Vanna AI Comprehensive Documentation (Expanded)

A single, detailed reference that consolidates the Vanna AI documentation topics you linked, and expands them with practical implementation details, examples, and code snippets derived from the open source Python package.

Build target: vanna (PyPI) 2.0.1 Generated on 2025-12-31 (Asia/Dubai)

Scope of this document

Covers the full set of placeholder documentation topics (auth, Web UI, deployment, training, tools, workflows, charts, memory backends, enhancers, system prompts, UI features, hooks, middlewares, error recovery, context enrichers, conversation filters, observability, audit logging, Python reference).

Adds pragmatic guidance and example code based on the publicly available Vanna Python package contents and its included examples.

Includes suggested best practices for production (security, performance, and operability).

Important notes

Some of the linked Vanna web docs are labeled as placeholders. Where the web docs are sparse, this document uses the package s actual implementation as the source of truth for behavior and APIs.

Code shown is based on the MIT licensed Vanna package; keep the license notice when copying substantial portions into your own materials.

Table of contents

1. Core Concepts and Architecture

Vanna (v2) is a modular agent framework. You compose an agent by selecting:

An LLM service (OpenAI, Anthropic, Ollama, Gemini, Azure OpenAI, or a mock for testing).

A tool registry containing built in tools (SQL, visualization, Python, file system, memory) and any custom tools you add.

A conversation store to persist the chat transcript and tool calls (memory or file based stores are available out of the box).

A user resolver that converts an incoming request context (headers/cookies/remote address) into a User object (used for authorization, scoping, audit, and memory).

Optional middlewares, lifecycle hooks, workflow handlers, context enrichers, and observability/audit components.

Key building blocks

Agent: orchestrates message processing, tool selection, and streaming UI components.

LlmService: provider adapter that sends requests / receives streamed tokens and tool calls.

Tool: structured function callable by the LLM. Tools declare a pydantic args schema and return a ToolResult (including optional UI component).

ToolRegistry: holds tool schemas and enforces access rules (e.g., group memberships) before execution.

ConversationStore: persists conversations and messages (including tool messages).

UserResolver: turns an incoming RequestContext into a User; also creates RequestContext objects for local use.

A typical high-level flow

1) Incoming message + RequestContext UserResolver.resolve_user() User

2) Agent builds LLM request: user message + conversation history + system prompt + tool schemas

3) LLM response streams back Agent yields UI components to the caller (server or REPL)

4) If the LLM requests a tool call ToolRegistry executes tool ToolResult is appended to conversation Agent continues

5) Optional: save memories, emit audit events, record observability spans, apply recovery strategies

Minimal agent (from the packaged examples)

This shows the smallest practical setup: an LLM service, a conversation store, a user resolver, and an agent configuration.

```
"""Minimal Claude + SQLite example ready for FastAPI."""
```

```
from __future__ import annotations

import os
from pathlib import Path

from vanna import AgentConfig, Agent
from vanna.core.registry import ToolRegistry
from vanna.integrations.anthropic import AnthropicLlmService
from vanna.integrations.sqlite import SqliteRunner
from vanna.integrations.local import LocalFileSystem
from vanna.tools import (
    RunSqlTool,
    # Visualization
    VisualizeDataTool,
    # Python execution
    RunPythonFileTool,
    PipInstallTool,
    # File system (for coding agents)
    SearchFilesTool,
    ListFilesTool,
    ReadFileTool,
    WriteFileTool,
)

_DB = Path(__file__).resolve().parents[2] / "Chinook.sqlite"


def create_demo_agent() -> Agent:
    # Load environment variables from .env file
    from dotenv import load_dotenv

    load_dotenv()

    llm = AnthropicLlmService(model=os.getenv("ANTHROPIC_MODEL", "claude-sonnet-4-5"))

    # Shared file system for all tools
    file_system = LocalFileSystem("./claude_data")

    tools = ToolRegistry()

    # 1. Basic SQL agent - query databases
    tools.register(
        RunSqlTool(
            sql_runner=SqliteRunner(database_path=str(_DB)),
            file_system=file_system,
        )
    )

    # 2. Add visualization - create charts from data
    tools.register(VisualizeDataTool(file_system=file_system))

    # 3. Add Python execution - build dashboards with artifacts
    # tools.register(RunPythonFileTool(file_system=file_system))
    # tools.register(PipInstallTool(file_system=file_system))

    # 4. Full coding agent - read, write, search files
    # tools.register(SearchFilesTool(file_system=file_system))
    # tools.register(ListFilesTool(file_system=file_system))
    # tools.register(ReadFileTool(file_system=file_system))
    # tools.register(WriteFileTool(file_system=file_system))

    return Agent(
        llm_service=llm,
        tool_registry=tools,
    )
```

2. Installation and Environment Setup

Python requirements

Python >= 3.9 (per package metadata).

Recommended: create a virtual environment per project (venv/conda).

Install the core package

```
pip install vanna
```

Install optional extras

Vanna uses extras to pull integrations and servers. Common choices:

Servers: pip install "vanna[servers]" (includes Flask + FastAPI extras).

OpenAI: pip install "vanna[openai]"

Anthropic: pip install "vanna[anthropic]"

Ollama: pip install "vanna[ollama]"

Database runners: pip install "vanna[postgres]", "vanna[mysql]", "vanna[snowflake]", etc.

Vector memory backends: pip install "vanna[chromadb]", "vanna[qdrant]", "vanna[pinecone]", etc.

Everything: pip install "vanna[all]" (largest dependency set).

Environment variables (typical)

OpenAI: OPENAI_API_KEY, OPENAI_MODEL (if supported by your selected integration).

Anthropic: ANTHROPIC_API_KEY, ANTHROPIC_MODEL (defaults to claude-sonnet-4-5 in the package implementation), ANTHROPIC_BASE_URL (optional).

Ollama: typically OLLAMA_HOST / base URL (see the Ollama integration).

Database runner credentials: use your DB client/driver conventions (e.g., POSTGRES_DSN, MYSQL_DSN) or pass connection strings directly in code.

Quick sanity check

```
python -c "import vanna; print('vanna imported OK')"
```

3. LLM Providers and Configuration

Vanna talks to an LLM through the LlmService interface. The package includes several ready-made implementations:

LLM service class	Integration module (package path)	Notes (typical use)
AnthropicLlmService	integrations/anthropic/llm.py	Anthropic Messages API wrapper. Use ANTHROPIC_API_K
AzureOpenAILlmService	integrations/azureopenai/llm.py	Azure OpenAI wrapper. Uses Azure endpoint + credentials.
GeminiLlmService	integrations/google/gemini.py	Google Gemini integration via google-genai/google-generativ
MockLlmService	integrations/mock/llm.py	Deterministic test LLM used in examples/tests.
OllamaLlmService	integrations/ollama/llm.py	Local Ollama HTTP integration (good for self-hosted models
OpenAILlmService	integrations/openai/llm.py	OpenAI Chat/Responses API wrapper. Use OPENAI_API_K

Anthropic configuration (matches the linked configure/anthropic/other topic)

The Anthropic integration reads model/key/base URL from constructor arguments or environment variables. A minimal setup looks like:

```
from vanna.integrations.anthropic import AnthropicLlmService

llm = AnthropicLlmService(
    api_key="YOUR_ANTHROPIC_API_KEY",
    model="claude-sonnet-4-5",  # or another supported Anthropic model
)
```

Reference implementation excerpt (AnthropicLlmService initializer)

```
"""
Anthropic LLM service implementation.

Implements the LlmService interface using Anthropic's Messages API
(anthropic>=0.8.0). Supports non-streaming and streaming text output.
Tool-calls (tool_use blocks) are surfaced at the end of a stream or after a
non-streaming call as ToolCall entries.
"""

from __future__ import annotations

import logging
import os
from typing import Any, AsyncGenerator, Dict, List, Optional, Tuple

logger = logging.getLogger(__name__)

from vanna.core.llm import (
    LlmService,
    LlmRequest,
    LlmResponse,
    LlmStreamChunk,
)
from vanna.core.tool import ToolCall, ToolSchema


class AnthropicLlmService(LlmService):
    """Anthropic Messages-backed LLM service.

    Args:
        model: Anthropic model name (e.g., "claude-sonnet-4-5", "claude-opus-4").
```

```
            Defaults to "claude-sonnet-4-5". Can also be set via ANTHROPIC_MODEL env var.
        api_key: API key; falls back to env `ANTHROPIC_API_KEY`.
        base_url: Optional custom base URL; env `ANTHROPIC_BASE_URL` if unset.
        extra_client_kwargs: Extra kwargs forwarded to `anthropic.Anthropic()`.
    """

    def __init__(
        self,
        model: Optional[str] = None,
        api_key: Optional[str] = None,
        base_url: Optional[str] = None,
        **extra_client_kwargs: Any,
    ) -> None:
        try:
            import anthropic
        except Exception as e:  # pragma: no cover
            raise ImportError(
                "anthropic package is required. Install with: pip install 'vanna[anthropic]'"
            ) from e

        # Model selection - use environment variable or default
        self.model = model or os.getenv("ANTHROPIC_MODEL", "claude-sonnet-4-5")
        api_key = api_key or os.getenv("ANTHROPIC_API_KEY")
        base_url = base_url or os.getenv("ANTHROPIC_BASE_URL")

        client_kwargs: Dict[str, Any] = {**extra_client_kwargs}
        if api_key:
            client_kwargs["api_key"] = api_key
        if base_url:
            client_kwargs["base_url"] = base_url

        self._client = anthropic.Anthropic(**client_kwargs)

    async def send_request(self, request: LlmRequest) -> LlmResponse:
        """Send a non-streaming request to Anthropic and return the response."""
        payload = self._build_payload(request)

        resp = self._client.messages.create(**payload)
```

OpenAI configuration

```
from vanna.integrations.openai import OpenAILlmService

llm = OpenAILlmService(api_key="YOUR_OPENAI_API_KEY", model="gpt-4.1-mini")  # example model name
```

Ollama configuration

```
from vanna.integrations.ollama import OllamaLlmService

llm = OllamaLlmService(
    model="llama3.1",
    base_url="http://localhost:11434",
)
```

Azure OpenAI configuration (high-level)

```
from vanna.integrations.azureopenai import AzureOpenAILlmService

llm = AzureOpenAILlmService(
    api_key="YOUR_AZURE_OPENAI_KEY",
    endpoint="https://YOUR-RESOURCE.openai.azure.com/",
    deployment="YOUR_DEPLOYMENT_NAME",
    api_version="2024-xx-xx",
)
```

Testing with the mock LLM

```
from vanna.integrations.mock import MockLlmService

llm = MockLlmService(response_content="Hello from a deterministic test LLM!")
```

4. SQL Execution and Database Integration

For analytics agents, the most common pattern is:

Use an LLM to translate a natural-language request into SQL.

Run SQL against a database using a database-specific SqlRunner integration.

Optionally visualize the resulting DataFrame using a chart tool.

Optionally save successful queries/tool-uses in agent memory for better future performance.

Database runners included in the package

SQL runner class	Integration module (package path)	Notes
BigQueryRunner	integrations/bigquery/sql_runner.py	Implements SqlRunner for the specified database engine.
ClickHouseRunner	integrations/clickhouse/sql_runner.py	Implements SqlRunner for the specified database engine.
DuckDBRunner	integrations/duckdb/sql_runner.py	Implements SqlRunner for the specified database engine.
HiveRunner	integrations/hive/sql_runner.py	Implements SqlRunner for the specified database engine.
MSSQLRunner	integrations/mssql/sql_runner.py	Implements SqlRunner for the specified database engine.
MySQLRunner	integrations/mysql/sql_runner.py	Implements SqlRunner for the specified database engine.
OracleRunner	integrations/oracle/sql_runner.py	Implements SqlRunner for the specified database engine.
PostgresRunner	integrations/postgres/sql_runner.py	Implements SqlRunner for the specified database engine.
PrestoRunner	integrations/presto/sql_runner.py	Implements SqlRunner for the specified database engine.
SnowflakeRunner	integrations/snowflake/sql_runner.py	Implements SqlRunner for the specified database engine.
SqliteRunner	integrations/sqlite/sql_runner.py	Implements SqlRunner for the specified database engine.

The built-in RunSqlTool

RunSqlTool is the standard execute SQL and return results tool. It supports returning both LLM-readable output and UI components (table, status cards).

Key behaviors (from the implementation)

Uses a SqlRunner to execute SQL and return a pandas DataFrame.

Enforces limits: max rows, max returned columns, and max SQL length (to keep responses safe and performant).

Can produce a rich DataFrame component for the UI.

Optional: includes query formatting and additional metadata in the UI component.

RunSqlTool excerpt (args + execution outline)

```
"""Generic SQL query execution tool with dependency injection."""

from typing import Any, Dict, List, Optional, Type, cast
import uuid
from vanna.core.tool import Tool, ToolContext, ToolResult
from vanna.components import (
    UiComponent,
    DataFrameComponent,
```

```
    NotificationComponent,
    ComponentType,
    SimpleTextComponent,
)
from vanna.capabilities.sql_runner import SqlRunner, RunSqlToolArgs
from vanna.capabilities.file_system import FileSystem
from vanna.integrations.local import LocalFileSystem


class RunSqlTool(Tool[RunSqlToolArgs]):
    """Tool that executes SQL queries using an injected SqlRunner implementation."""

    def __init__(
        self,
        sql_runner: SqlRunner,
        file_system: Optional[FileSystem] = None,
        custom_tool_name: Optional[str] = None,
        custom_tool_description: Optional[str] = None,
    ):
        """Initialize the tool with a SqlRunner implementation.

        Args:
            sql_runner: SqlRunner implementation that handles actual query execution
            file_system: FileSystem implementation for saving results (defaults to LocalFileSystem)
            custom_tool_name: Optional custom name for the tool (overrides default "run_sql")
            custom_tool_description: Optional custom description for the tool (overrides default descrip
        """
        self.sql_runner = sql_runner
        self.file_system = file_system or LocalFileSystem()
        self._custom_name = custom_tool_name
        self._custom_description = custom_tool_description

    @property
    def name(self) -> str:
        return self._custom_name if self._custom_name else "run_sql"

    @property
    def description(self) -> str:
        return (
            self._custom_description
            if self._custom_description
            else "Execute SQL queries against the configured database"
        )

    def get_args_schema(self) -> Type[RunSqlToolArgs]:
        return RunSqlToolArgs

    async def execute(self, context: ToolContext, args: RunSqlToolArgs) -> ToolResult:
        """Execute a SQL query using the injected SqlRunner."""
        try:
            # Use the injected SqlRunner to execute the query
            df = await self.sql_runner.run_sql(args, context)

            # Determine query type
            query_type = args.sql.strip().upper().split()[0]

            if query_type == "SELECT":
                # Handle SELECT queries with results
                if df.empty:
                    result = "Query executed successfully. No rows returned."
                    ui_component = UiComponent(
                        rich_component=DataFrameComponent(
                            rows=[],
                            columns=[],
                            title="Query Results",
                            description="No rows returned",
                        ),
                        simple_component=SimpleTextComponent(text=result),
```

```
                )
                metadata = {
                    "row_count": 0,
                    "columns": [],
                    "query_type": query_type,
                    "results": [],
                }
            else:
                # Convert DataFrame to records
                results_data = df.to_dict("records")
                columns = df.columns.tolist()
                row_count = len(df)

                # Write DataFrame to CSV file for downstream tools
                file_id = str(uuid.uuid4())[:8]
                filename = f"query_results_{file_id}.csv"
                csv_content = df.to_csv(index=False)
                await self.file_system.write_file(
                    filename, csv_content, context, overwrite=True
                )

                # Create result text for LLM with truncated results
                results_preview = csv_content
                if len(results_preview) > 1000:
                    results_preview = (
                        results_preview[:1000]
                        + "\n(Results truncated to 1000 characters. FOR LARGE RESULTS YOU DO NOT NEE
                    )

                result = f"{results_preview}\n\nResults saved to file: {filename}\n\n**IMPORTANT: FO

                # Create DataFrame component for UI
                dataframe_component = DataFrameComponent.from_records(
                    records=cast(List[Dict[str, Any]], results_data),
                    title="Query Results",
                    description=f"SQL query returned {row_count} rows with {len(columns)} columns",
                )

                ui_component = UiComponent(
                    rich_component=dataframe_component,
                    simple_component=SimpleTextComponent(text=result),
                )

                metadata = {
                    "row_count": row_count,
                    "columns": columns,
                    "query_type": query_type,
                    "results": results_data,
                    "output_file": filename,
                }
        else:
            # For non-SELECT queries (INSERT, UPDATE, DELETE, etc.)
            # The SqlRunner should return a DataFrame with affected row count
            rows_affected = len(df) if not df.empty else 0
            result = (
                f"Query executed successfully. {rows_affected} row(s) affected."
            )

            metadata = {"rows_affected": rows_affected, "query_type": query_type}
            ui_component = UiComponent(
                rich_component=NotificationComponent(
                    type=ComponentType.NOTIFICATION, level="success", message=result
                ),
                simple_component=SimpleTextComponent(text=result),
            )

        return ToolResult(
            success=True,
```

```
                result_for_llm=result,
                ui_component=ui_component,
                metadata=metadata,
            )

        except Exception as e:
            error_message = f"Error executing query: {str(e)}"
            return ToolResult(
                success=False,
                result_for_llm=error_message,
                ui_component=UiComponent(
                    rich_component=NotificationComponent(
                        type=ComponentType.NOTIFICATION,
                        level="error",
                        message=error_message,
                    ),
                    simple_component=SimpleTextComponent(text=error_message),
                ),
                error=str(e),
                metadata={"error_type": "sql_error"},
            )
```

Example: Postgres runner + RunSqlTool wired into an agent

```
from vanna import Agent, AgentConfig
from vanna.core.registry import ToolRegistry
from vanna.integrations.openai import OpenAILlmService
from vanna.integrations.postgres import PostgresRunner
from vanna.integrations.local import MemoryConversationStore
from vanna.core.user import SimpleUserResolver
from vanna.tools import RunSqlTool

llm = OpenAILlmService(api_key="YOUR_OPENAI_API_KEY")
runner = PostgresRunner(connection_string="postgresql://user:pass@host:5432/dbname")

tools = ToolRegistry()
tools.register(RunSqlTool(sql_runner=runner))

agent = Agent(
    llm_service=llm,
    tool_registry=tools,
    user_resolver=SimpleUserResolver(),
    conversation_store=MemoryConversationStore(),
    config=AgentConfig(stream_responses=True),
)
```

5. Training and Memory Backends

In Vanna v2, training typically means saving and reusing knowledge about what tools worked for similar questions (tool use memory), plus optional free text memory.

Conceptually:

Tool memories: For questions like X, the correct tool call was run_sql({sql: ...}) and it succeeded.

Text memories: Remember this user preference / rule / project note.

Search: fetch similar memories and inject them into the system prompt (or provide as context to the LLM).

AgentMemory interface

```
"""
Agent memory capability interface for tool usage learning.

This module contains the abstract base class for agent memory operations,
following the same pattern as the FileSystem interface.
"""

from __future__ import annotations

from abc import ABC, abstractmethod
from typing import TYPE_CHECKING, Any, Dict, List, Optional

if TYPE_CHECKING:
    from vanna.core.tool import ToolContext
    from .models import (
        ToolMemorySearchResult,
        TextMemory,
        TextMemorySearchResult,
        ToolMemory,
    )


class AgentMemory(ABC):
    """Abstract base class for agent memory operations."""

    @abstractmethod
    async def save_tool_usage(
        self,
        question: str,
        tool_name: str,
        args: Dict[str, Any],
        context: "ToolContext",
        success: bool = True,
        metadata: Optional[Dict[str, Any]] = None,
    ) -> None:
        """Save a tool usage pattern for future reference."""
        pass

    @abstractmethod
    async def save_text_memory(
        self, content: str, context: "ToolContext"
    ) -> "TextMemory":
        """Save a free-form text memory."""
        pass

    @abstractmethod
    async def search_similar_usage(
        self,
```

```
        question: str,
        context: "ToolContext",
        *,
        limit: int = 10,
        similarity_threshold: float = 0.7,
        tool_name_filter: Optional[str] = None,
    ) -> List[ToolMemorySearchResult]:
        """Search for similar tool usage patterns based on a question."""
        pass

    @abstractmethod
    async def search_text_memories(
        self,
        query: str,
        context: "ToolContext",
        *,
        limit: int = 10,
        similarity_threshold: float = 0.7,
    ) -> List["TextMemorySearchResult"]:
        """Search stored text memories based on a query."""
        pass

    @abstractmethod
    async def get_recent_memories(
        self, context: "ToolContext", limit: int = 10
    ) -> List[ToolMemory]:
        """Get recently added memories. Returns most recent memories first."""
        pass

    @abstractmethod
    async def get_recent_text_memories(
        self, context: "ToolContext", limit: int = 10
    ) -> List["TextMemory"]:
        """Fetch recently stored text memories."""
        pass

    @abstractmethod
    async def delete_by_id(self, context: "ToolContext", memory_id: str) -> bool:
        """Delete a memory by its ID. Returns True if deleted, False if not found."""
        pass

    @abstractmethod
    async def delete_text_memory(self, context: "ToolContext", memory_id: str) -> bool:
        """Delete a text memory by its ID. Returns True if deleted, False if not found."""
        pass

    @abstractmethod
    async def clear_memories(
        self,
        context: "ToolContext",
        tool_name: Optional[str] = None,
        before_date: Optional[str] = None,
    ) -> int:
        """Clear stored memories (tool or text). Returns number of memories deleted."""
        pass
```

Memory backends included in the package

AgentMemory backend	Integration module (package path)	Notes
AzureAISearchAgentMemory	integrations/azuresearch/agent_memory.py	Vector / search-based memory store for tool usage & text me
ChromaAgentMemory	integrations/chromadb/agent_memory.py	Vector / search-based memory store for tool usage & text me
FAISSAgentMemory	integrations/faiss/agent_memory.py	Vector / search-based memory store for tool usage & text me
MarqoAgentMemory	integrations/marqo/agent_memory.py	Vector / search-based memory store for tool usage & text me

MilvusAgentMemory	integrations/milvus/agent_memory.py	Vector / search-based memory store for tool usage & text me
OpenSearchAgentMemory	integrations/opensearch/agent_memory.py	Vector / search-based memory store for tool usage & text me
PineconeAgentMemory	integrations/pinecone/agent_memory.py	Vector / search-based memory store for tool usage & text me
QdrantAgentMemory	integrations/qdrant/agent_memory.py	Vector / search-based memory store for tool usage & text me
WeaviateAgentMemory	integrations/weaviate/agent_memory.py	Vector / search-based memory store for tool usage & text me

Built-in memory tools

SaveQuestionToolArgsTool stores a successful tool call pattern.

SearchSavedCorrectToolUsesTool retrieves similar historical tool uses.

SaveTextMemoryTool stores arbitrary text memory for later retrieval.

Memory tool excerpt (search + save patterns)

```
"""
Agent memory tools.

This module provides agent memory operations through an abstract AgentMemory interface,
allowing for different implementations (local vector DB, remote cloud service, etc.).
The tools access AgentMemory via ToolContext, which is populated by the Agent.
"""

import logging
from typing import Any, Dict, List, Optional, Type
from pydantic import BaseModel, Field

logger = logging.getLogger(__name__)

from vanna.core.tool import Tool, ToolContext, ToolResult
from vanna.core.agent.config import UiFeature
from vanna.capabilities.agent_memory import AgentMemory
from vanna.components import (
    UiComponent,
    StatusBarUpdateComponent,
    CardComponent,
)


class SaveQuestionToolArgsParams(BaseModel):
    """Parameters for saving question-tool-argument combinations."""

    question: str = Field(description="The original question that was asked")
    tool_name: str = Field(
        description="The name of the tool that was used successfully"
    )
    args: Dict[str, Any] = Field(
        description="The arguments that were passed to the tool"
    )


class SearchSavedCorrectToolUsesParams(BaseModel):
    """Parameters for searching saved tool usage patterns."""

    question: str = Field(
        description="The question to find similar tool usage patterns for"
    )
    limit: Optional[int] = Field(
        default=10, description="Maximum number of results to return"
    )
    similarity_threshold: Optional[float] = Field(
        default=0.7, description="Minimum similarity score for results (0.0-1.0)"
```

```
    )
    tool_name_filter: Optional[str] = Field(
        default=None, description="Filter results to specific tool name"
    )


class SaveTextMemoryParams(BaseModel):
    """Parameters for saving free-form text memories."""

    content: str = Field(description="The text content to save as a memory")


class SaveQuestionToolArgsTool(Tool[SaveQuestionToolArgsParams]):
    """Tool for saving successful question-tool-argument combinations."""

    @property
    def name(self) -> str:
        return "save_question_tool_args"

    @property
    def description(self) -> str:
        return (
            "Save a successful question-tool-argument combination for future reference"
        )

    def get_args_schema(self) -> Type[SaveQuestionToolArgsParams]:
        return SaveQuestionToolArgsParams

    async def execute(
        self, context: ToolContext, args: SaveQuestionToolArgsParams
    ) -> ToolResult:
        """Save the tool usage pattern to agent memory."""
        try:
            await context.agent_memory.save_tool_usage(
                question=args.question,
                tool_name=args.tool_name,
                args=args.args,
                context=context,
                success=True,
            )

            success_msg = (
                f"Successfully saved usage pattern for '{args.tool_name}' tool"
            )
            return ToolResult(
                success=True,
                result_for_llm=success_msg,
                ui_component=UiComponent(
                    rich_component=StatusBarUpdateComponent(
                        status="success",
                        message="Saved to memory",
                        detail=f"Saved pattern for '{args.tool_name}'",
                    ),
                    simple_component=None,
                ),
            )

        except Exception as e:
            error_message = f"Failed to save memory: {str(e)}"
            return ToolResult(
                success=False,
                result_for_llm=error_message,
                ui_component=UiComponent(
                    rich_component=StatusBarUpdateComponent(
                        status="error", message="Failed to save memory", detail=str(e)
                    ),
                    simple_component=None,
                ),
```

```
                error=str(e),
            )


class SearchSavedCorrectToolUsesTool(Tool[SearchSavedCorrectToolUsesParams]):
    """Tool for searching saved tool usage patterns."""

    @property
    def name(self) -> str:
        return "search_saved_correct_tool_uses"

    @property
    def description(self) -> str:
        return "Search for similar tool usage patterns based on a question"

    def get_args_schema(self) -> Type[SearchSavedCorrectToolUsesParams]:
        return SearchSavedCorrectToolUsesParams

    async def execute(
        self, context: ToolContext, args: SearchSavedCorrectToolUsesParams
    ) -> ToolResult:
        """Search for similar tool usage patterns."""
        try:
            results = await context.agent_memory.search_similar_usage(
                question=args.question,
                context=context,
                limit=args.limit or 10,
                similarity_threshold=args.similarity_threshold or 0.7,
                tool_name_filter=args.tool_name_filter,
            )

            if not results:
                no_results_msg = (
                    "No similar tool usage patterns found for this question."
                )

                # Check if user has access to detailed memory results
                ui_features_available = context.metadata.get(
                    "ui_features_available", []
                )
                show_detailed_results = (
                    UiFeature.UI_FEATURE_SHOW_MEMORY_DETAILED_RESULTS
                    in ui_features_available
                )

                # Create UI component based on access level
                if show_detailed_results:
                    # Admin view: Show card indicating 0 results
                    ui_component = UiComponent(
                        rich_component=CardComponent(
                            title="  Memory Search: 0 Results",
                            content="No similar tool usage patterns found for this question.\n\nSearched
                            icon=" ",
                            status="info",
                            collapsible=True,
                            collapsed=True,
                            markdown=True,
                        ),
                        simple_component=None,
                    )
                else:
                    # Non-admin view: Simple status message
                    ui_component = UiComponent(
                        rich_component=StatusBarUpdateComponent(
                            status="idle",
                            message="No similar patterns found",
                            detail="Searched agent memory",
                        ),
```

```
                    simple_component=None,
                )

            return ToolResult(
                success=True,
                result_for_llm=no_results_msg,
                ui_component=ui_component,
            )

        # Format results for LLM
        results_text = f"Found {len(results)} similar tool usage pattern(s):\n\n"
        for i, result in enumerate(results, 1):
            memory = result.memory
            results_text += f"{i}. {memory.tool_name} (similarity: {result.similarity_score:.2f})\n"
            results_text += f"   Question: {memory.question}\n"
            results_text += f"   Args: {memory.args}\n\n"

        logger.info(f"Agent memory search results: {results_text.strip()}")

        # Check if user has access to detailed memory results
        ui_features_available = context.metadata.get("ui_features_available", [])
        show_detailed_results = (
            UiFeature.UI_FEATURE_SHOW_MEMORY_DETAILED_RESULTS
            in ui_features_available
        )

        # Create UI component based on access level
        if show_detailed_results:
            # Admin view: Show detailed results in collapsible card
            detailed_content = "**Retrieved memories passed to LLM:**\n\n"
            for i, result in enumerate(results, 1):
                memory = result.memory
                detailed_content += f"**{i}. {memory.tool_name}** (similarity: {result.similarity_sc
                detailed_content += f"- **Question:** {memory.question}\n"
                detailed_content += f"- **Arguments:** `{memory.args}`\n"
                if memory.timestamp:
                    detailed_content += f"- **Timestamp:** {memory.timestamp}\n"
```

Example: enable memory with Chroma (local vector DB)

```
from vanna import Agent, AgentConfig
from vanna.core.registry import ToolRegistry
from vanna.core.user import SimpleUserResolver
from vanna.integrations.local import MemoryConversationStore
from vanna.integrations.openai import OpenAILlmService
from vanna.integrations.chromadb import ChromaAgentMemory
from vanna.tools import (
    SaveQuestionToolArgsTool,
    SearchSavedCorrectToolUsesTool,
    SaveTextMemoryTool,
)

memory = ChromaAgentMemory(persist_directory="./chroma_memory")

tools = ToolRegistry()
tools.register(SaveQuestionToolArgsTool(agent_memory=memory))
tools.register(SearchSavedCorrectToolUsesTool(agent_memory=memory))
tools.register(SaveTextMemoryTool(agent_memory=memory))

agent = Agent(
    llm_service=OpenAILlmService(api_key="..."),
    tool_registry=tools,
    user_resolver=SimpleUserResolver(),
    conversation_store=MemoryConversationStore(),
    config=AgentConfig(stream_responses=True),
)
```

6. Tools

Tools are the main extension point. A tool:

Defines a unique name and a natural-language description used for tool selection.

Provides a pydantic args schema (so the LLM can call it safely with typed arguments).

Executes with a ToolContext (which includes the resolved user, request metadata, and other shared context).

Returns a ToolResult containing: success flag, LLM-visible result, optional UI component, and optional error details.

Built-in tools shipped in the package

Tool class	Module (package path)	Typical purpose
EditFileTool	tools/file_system.py	Edit a file via patch/diff.
ListFilesTool	tools/file_system.py	List directory contents in configured root.
PipInstallTool	tools/python.py	Install Python packages (for coding agents).
ReadFileTool	tools/file_system.py	Read a file from the configured root.
RunPythonFileTool	tools/python.py	Execute a Python file (controlled).
RunSqlTool	tools/run_sql.py	Run SQL using a configured SqlRunner; returns DataFrame
SaveQuestionToolArgsTool	tools/agent_memory.py	Record successful tool call patterns.
SaveTextMemoryTool	tools/agent_memory.py	Save arbitrary text memory.
SearchFilesTool	tools/file_system.py	Search for patterns in files.
SearchSavedCorrectToolUsesTool	tools/agent_memory.py	Retrieve similar successful patterns.
VisualizeDataTool	tools/visualize_data.py	Generate a chart from a pandas DataFrame (Plotly).
WriteFileTool	tools/file_system.py	Write/overwrite a file in configured root.

Tool base class and ToolResult (excerpt)

```
"""
Tool domain interface.

This module contains the abstract base class for tools.
"""

from abc import ABC, abstractmethod
from typing import Generic, List, Type, TypeVar

from .models import ToolContext, ToolResult, ToolSchema

# Type variable for tool argument types
T = TypeVar("T")


class Tool(ABC, Generic[T]):
    """Abstract base class for tools."""

    @property
    @abstractmethod
    def name(self) -> str:
```

```
        """Unique name for this tool."""
        pass

    @property
    @abstractmethod
    def description(self) -> str:
        """Description of what this tool does."""
        pass

    @property
    def access_groups(self) -> List[str]:
        """Groups permitted to access this tool."""
        return []

    @abstractmethod
    def get_args_schema(self) -> Type[T]:
        """Return the Pydantic model for arguments."""
        pass

    @abstractmethod
    async def execute(self, context: ToolContext, args: T) -> ToolResult:
        """Execute the tool with validated arguments.

        Args:
            context: Execution context containing user, conversation_id, and request_id
            args: Validated tool arguments

        Returns:
            ToolResult with success status, result for LLM, and optional UI component
        """
        pass

    def get_schema(self) -> ToolSchema:
        """Generate tool schema for LLM."""
        from typing import Any, cast

        args_model = self.get_args_schema()
        # Get the schema - args_model should be a Pydantic model class
        schema = (
            cast(Any, args_model).model_json_schema()
            if hasattr(args_model, "model_json_schema")
            else {}
        )
        return ToolSchema(
            name=self.name,
            description=self.description,
            parameters=schema,
            access_groups=self.access_groups,
        )
```

Creating a custom tool

A custom tool is typically a small class that subclasses Tool[ArgsModel]. The example below is a safe hello world tool.

```
from typing import Type
from pydantic import BaseModel, Field
from vanna.core.tool import Tool, ToolContext, ToolResult

class GreetArgs(BaseModel):
    name: str = Field(description="Person to greet")

class GreetTool(Tool[GreetArgs]):
    @property
    def name(self) -> str:
        return "greet"
```

```
    @property
    def description(self) -> str:
        return "Greet a person by name."

    def get_args_schema(self) -> Type[GreetArgs]:
        return GreetArgs

    async def execute(self, context: ToolContext, args: GreetArgs) -> ToolResult:
        return ToolResult(
            success=True,
            result_for_llm=f"Hello, {args.name}!",
        )
```

Registering tools and enforcing permissions

Tools may specify required group memberships. ToolRegistry checks access before execution.

```
"""
Tool registry for the Vanna Agents framework.

This module provides the ToolRegistry class for managing and executing tools.
"""

import time
from typing import TYPE_CHECKING, Any, Dict, List, Optional, Type, TypeVar, Union

from .tool import Tool, ToolCall, ToolContext, ToolRejection, ToolResult, ToolSchema
from .user import User

if TYPE_CHECKING:
    from .audit import AuditLogger
    from .agent.config import AuditConfig

T = TypeVar("T")


class _LocalToolWrapper(Tool[T]):
    """Wrapper for tools with configurable access groups."""

    def __init__(self, wrapped_tool: Tool[T], access_groups: List[str]):
        self._wrapped_tool = wrapped_tool
        self._access_groups = access_groups

    @property
    def name(self) -> str:
        return self._wrapped_tool.name

    @property
    def description(self) -> str:
        return self._wrapped_tool.description

    @property
    def access_groups(self) -> List[str]:
        return self._access_groups

    def get_args_schema(self) -> Type[T]:
        return self._wrapped_tool.get_args_schema()

    async def execute(self, context: ToolContext, args: T) -> ToolResult:
        return await self._wrapped_tool.execute(context, args)


class ToolRegistry:
    """Registry for managing tools."""

    def __init__(
```

```
        self,
        audit_logger: Optional["AuditLogger"] = None,
        audit_config: Optional["AuditConfig"] = None,
    ) -> None:
        self._tools: Dict[str, Tool[Any]] = {}
        self.audit_logger = audit_logger
        if audit_config is not None:
            self.audit_config = audit_config
        else:
            from .agent.config import AuditConfig

            self.audit_config = AuditConfig()

    def register_local_tool(self, tool: Tool[Any], access_groups: List[str]) -> None:
        """Register a local tool with optional access group restrictions.

        Args:
            tool: The tool to register
            access_groups: List of groups that can access this tool.
                           If None or empty, tool is accessible to all users.
        """
        if tool.name in self._tools:
            raise ValueError(f"Tool '{tool.name}' already registered")

        if access_groups:
            # Wrap the tool with access groups
            wrapped_tool = _LocalToolWrapper(tool, access_groups)
            self._tools[tool.name] = wrapped_tool
        else:
            # No access restrictions, register as-is
            self._tools[tool.name] = tool

    async def get_tool(self, name: str) -> Optional[Tool[Any]]:
        """Get a tool by name."""
        return self._tools.get(name)

    async def list_tools(self) -> List[str]:
        """List all registered tool names."""
        return list(self._tools.keys())

    async def get_schemas(self, user: Optional[User] = None) -> List[ToolSchema]:
        """Get schemas for all tools accessible to user."""
        schemas = []
        for tool in self._tools.values():
            if user is None or await self._validate_tool_permissions(tool, user):
                schemas.append(tool.get_schema())
        return schemas

    async def _validate_tool_permissions(self, tool: Tool[Any], user: User) -> bool:
        """Validate if user has access to tool based on group membership.

        Checks for intersection between user's group memberships and tool's access groups.
        If tool has no access groups specified, it's accessible to all users.
        """
        tool_access_groups = tool.access_groups
        if not tool_access_groups:
            return True

        user_groups = set(user.group_memberships)
        tool_groups = set(tool_access_groups)
        # Grant access if any group in user.group_memberships exists in tool.access_groups
        return bool(user_groups & tool_groups)

    async def transform_args(
        self,
        tool: Tool[T],
        args: T,
        user: User,
```

```
        context: ToolContext,
    ) -> Union[T, ToolRejection]:
        """Transform and validate tool arguments based on user context.

        This method allows per-user transformation of tool arguments, such as:
        - Applying row-level security (RLS) to SQL queries
        - Filtering available options based on user permissions
        - Validating required arguments are present
        - Redacting sensitive fields

        The default implementation performs no transformation (NoOp).
        Subclasses can override this method to implement custom transformation logic.

        Args:
            tool: The tool being executed
            args: Already Pydantic-validated arguments
            user: The user executing the tool
            context: Full execution context

        Returns:
            Either:
            - Transformed arguments (may be unchanged if no transformation needed)
            - ToolRejection with explanation of why args were rejected
        """
        return args  # Default: no transformation (NoOp)

    async def execute(
        self,
        tool_call: ToolCall,
        context: ToolContext,
    ) -> ToolResult:
        """Execute a tool call with validation."""
        tool = await self.get_tool(tool_call.name)
        if not tool:
            msg = f"Tool '{tool_call.name}' not found"
            return ToolResult(
                success=False,
                result_for_llm=msg,
                ui_component=None,
                error=msg,
            )

        # Validate group access
        if not await self._validate_tool_permissions(tool, context.user):
            msg = f"Insufficient group access for tool '{tool_call.name}'"

            # Audit access denial
            if (
                self.audit_logger
                and self.audit_config
                and self.audit_config.log_tool_access_checks
            ):
                await self.audit_logger.log_tool_access_check(
                    user=context.user,
                    tool_name=tool_call.name,
                    access_granted=False,
                    required_groups=tool.access_groups,
                    context=context,
                    reason=msg,
                )

            return ToolResult(
                success=False,
                result_for_llm=msg,
                ui_component=None,
                error=msg,
            )
```

```
        # Validate and parse arguments
        try:
            args_model = tool.get_args_schema()
            validated_args = args_model.model_validate(tool_call.arguments)
        except Exception as e:
            msg = f"Invalid arguments: {str(e)}"
            return ToolResult(
                success=False,
                result_for_llm=msg,
                ui_component=None,
                error=msg,
            )

        # Transform/validate arguments based on user context
        transform_result = await self.transform_args(
            tool=tool,
            args=validated_args,
            user=context.user,
            context=context,
        )

        if isinstance(transform_result, ToolRejection):
            return ToolResult(
                success=False,
                result_for_llm=transform_result.reason,
                ui_component=None,
                error=transform_result.reason,
            )

        # Use transformed arguments for execution
        final_args = transform_result

        # Audit successful access check
        if (
            self.audit_logger
```

7. Workflow Handlers

Workflow handlers control the higher-level conversation workflow around an agent: starter UI, commands (e.g., /help), setup checks, and other policy-level logic that is not purely LLM-based.

WorkflowHandler interface (excerpt)

```
"""
Base workflow handler interface.

Workflow triggers allow you to execute deterministic workflows in response to
user messages before they are sent to the LLM. This is useful for:
- Command handling (e.g., /help, /reset)
- Pattern-based routing (e.g., report generation)
- State-based workflows (e.g., onboarding flows)
- Quota enforcement with custom responses
"""

from abc import ABC, abstractmethod
from typing import (
    TYPE_CHECKING,
    Optional,
    Union,
    List,
    AsyncGenerator,
    Callable,
    Awaitable,
)
from dataclasses import dataclass

if TYPE_CHECKING:
    from ..user.models import User
    from ..storage import Conversation
    from ...components import UiComponent
    from ..agent.agent import Agent


@dataclass
class WorkflowResult:
    """Result from a workflow handler attempt.

    When a workflow handles a message, it can optionally return UI components to stream
    to the user and/or mutate the conversation state.

    Attributes:
        should_skip_llm: If True, the workflow handled the message and LLM processing is skipped.
                         If False, the message continues to the agent/LLM.
        components: Optional UI components to stream back to the user.
                    Can be a list or async generator for streaming responses.
        conversation_mutation: Optional async callback to modify conversation state
                               (e.g., clearing messages, adding system events).

    Example:
        # Simple command response
        WorkflowResult(
            should_skip_llm=True,
            components=[RichTextComponent(content="Help text here")]
        )

        # With conversation mutation
        async def clear_history(conv):
            conv.messages.clear()

        WorkflowResult(
```

```
            should_skip_llm=True,
            components=[StatusCardComponent(...)],
            conversation_mutation=clear_history
        )

        # Not handled, continue to agent
        WorkflowResult(should_skip_llm=False)
    """

    should_skip_llm: bool
    components: Optional[
        Union[List["UiComponent"], AsyncGenerator["UiComponent", None]]
    ] = None
    conversation_mutation: Optional[Callable[["Conversation"], Awaitable[None]]] = None


class WorkflowHandler(ABC):
    """Base class for handling deterministic workflows before LLM processing.

    Implement this interface to intercept user messages and execute deterministic
    workflows instead of sending to the LLM. This is the first extensibility point
    in the agent's message processing pipeline, running after user resolution and
    conversation loading but before the message is added to conversation history
    or sent to the LLM.

    Use cases:
    - Slash commands (/help, /reset, /report)
    - Pattern-based routing (regex matching)
    - State-based workflows (onboarding, surveys)
    - Custom quota enforcement with helpful messages
    - Deterministic report generation
    - Starter UI (buttons, welcome messages) when conversation begins

    Example:
        class CommandWorkflow(WorkflowHandler):
            async def try_handle(self, agent, user, conversation, message):
                if message.startswith("/help"):
                    return WorkflowResult(
                        should_skip_llm=True,
                        components=[
                            RichTextComponent(
                                content="Available commands:\\n- /help\\n- /reset",
                                markdown=True
                            )
                        ]
                    )

                # Execute tool for reports
                if message.startswith("/report"):
                    tool = await agent.tool_registry.get_tool("generate_report")
                    result = await tool.execute(ToolContext(user=user), {})
                    return WorkflowResult(should_skip_llm=True, components=[result.ui_component])

                # Not handled, continue to agent
                return WorkflowResult(should_skip_llm=False)

            async def get_starter_ui(self, agent, user, conversation):
                return [
                    RichTextComponent(content=f"Welcome {user.username}!"),
                    ButtonComponent(label="Generate Report", value="/report"),
                ]

        agent = Agent(
            llm_service=...,
            tool_registry=...,
            user_resolver=...,
            workflow_handler=CommandWorkflow()
        )
```

```
    Observability:
        The agent automatically creates an "agent.workflow_handler" span when
        a WorkflowHandler is configured, allowing you to monitor handler
        performance and outcomes.
    """

    @abstractmethod
    async def try_handle(
        self, agent: "Agent", user: "User", conversation: "Conversation", message: str
    ) -> WorkflowResult:
        """Attempt to handle a workflow for the given message.

        This method is called for every user message before it reaches the LLM.
        Inspect the message content, user context, and conversation state to
        decide whether to execute a deterministic workflow or allow normal
        agent processing.

        Args:
            agent: The agent instance, providing access to tool_registry, config,
                   and observability_provider for tool execution and logging.
            user: The user who sent the message, including their ID, permissions,
                  and metadata. Use this for permission checks or personalization.
            conversation: The current conversation context, including message history.
                          Can be inspected for state-based workflows.
            message: The user's raw message content.

        Returns:
            WorkflowResult with should_skip_llm=True to execute a workflow and skip LLM,
            or should_skip_llm=False to continue normal agent processing.

            When should_skip_llm=True:
            - The message is NOT added to conversation history automatically
            - The components are streamed to the user
            - The conversation_mutation callback (if provided) is executed
            - The agent returns without calling the LLM

            When should_skip_llm=False:
            - The message is added to conversation history
            - Normal agent processing continues (LLM call, tool execution, etc.)

        Example:
            async def try_handle(self, agent, user, conversation, message):
                # Pattern matching with tool execution
                if message.startswith("/report"):
                    # Execute tool from registry
                    tool = await agent.tool_registry.get_tool("generate_sales_report")
                    context = ToolContext(user=user, conversation=conversation)
                    result = await tool.execute(context, {})

                    return WorkflowResult(
                        should_skip_llm=True,
                        components=[...]
                    )

                # State-based workflow
                if user.metadata.get("needs_onboarding"):
                    return await self._onboarding_flow(agent, user, message)

                # Permission check
                if message.startswith("/admin") and "admin" not in user.permissions:
                    return WorkflowResult(
                        should_skip_llm=True,
                        components=[RichTextComponent(content="Access denied.")]
                    )

                # Continue to agent
                return WorkflowResult(should_skip_llm=False)
        """
```

```
        pass

    async def get_starter_ui(
        self, agent: "Agent", user: "User", conversation: "Conversation"
    ) -> Optional[List["UiComponent"]]:
        """Provide UI components when a conversation starts.
```

DefaultWorkflowHandler (what it provides)

Starter UI when the front-end loads (can detect missing SQL or memory tools and suggest setup).

Built-in commands like /help and /status.

Basic health checks based on registered tools.

DefaultWorkflowHandler excerpt

```
"""
Default workflow handler implementation with setup health checking.

This module provides a default implementation of the WorkflowHandler interface
that provides a smart starter UI based on available tools and setup status.
"""

from typing import TYPE_CHECKING, List, Optional, Dict, Any
import traceback
import uuid
from .base import WorkflowHandler, WorkflowResult

if TYPE_CHECKING:
    from ..agent.agent import Agent
    from ..user.models import User
    from ..storage import Conversation

# Import components at module level to avoid circular imports
from vanna.components import (
    UiComponent,
    RichTextComponent,
    StatusCardComponent,
    ButtonComponent,
    ButtonGroupComponent,
    SimpleTextComponent,
    CardComponent,
)

# Note: StatusCardComponent and ButtonGroupComponent are kept for /status command compatibility


class DefaultWorkflowHandler(WorkflowHandler):
    """Default workflow handler that provides setup health checking and starter UI.

    This handler provides a starter UI that:
    - Checks if run_sql tool is available (critical)
    - Checks if memory tools are available (warning if missing)
    - Checks if visualization tools are available
    - Provides appropriate setup guidance based on what's missing
    """

    def __init__(self, welcome_message: Optional[str] = None):
        """Initialize with optional custom welcome message.

        Args:
            welcome_message: Optional custom welcome message. If not provided,
                             generates one based on available tools.
        """
        self.welcome_message = welcome_message
```

```
async def try_handle(
    self, agent: "Agent", user: "User", conversation: "Conversation", message: str
) -> WorkflowResult:
    """Handle basic commands, but mostly passes through to LLM."""

    # Handle basic help command
    if message.strip().lower() in ["/help", "help", "/h"]:
        # Check if user is admin
        is_admin = "admin" in user.group_memberships

        help_content = (
            "##   Vanna AI Assistant\n\n"
            "I'm your AI data analyst! Here's what I can help you with:\n\n"
            "**  Natural Language Queries**\n"
            '- "Show me sales data for last quarter"\n'
            '- "Which customers have the highest orders?"\n'
            '- "Create a chart of revenue by month"\n\n'
            "**  Commands**\n"
            "- `/help` - Show this help message\n"
        )

        if is_admin:
            help_content += (
                "\n**  Admin Commands**\n"
                "- `/status` - Check setup status\n"
                "- `/memories` - View and manage recent memories\n"
                "- `/delete [id]` - Delete a memory by ID\n"
            )

        help_content += "\n\nJust ask me anything about your data in plain English!"

        return WorkflowResult(
            should_skip_llm=True,
            components=[
                UiComponent(
                    rich_component=RichTextComponent(
                        content=help_content,
                        markdown=True,
                    ),
                    simple_component=None,
                )
            ],
        )

    # Handle status check command (admin-only)
    if message.strip().lower() in ["/status", "status"]:
        # Check if user is admin
        if "admin" not in user.group_memberships:
            return WorkflowResult(
                should_skip_llm=True,
                components=[
                    UiComponent(
                        rich_component=RichTextComponent(
                            content="#   Access Denied\n\n"
                            "The `/status` command is only available to administrators.\n\n"
                            "If you need access to system status information, please contact your sy
                            markdown=True,
                        ),
                        simple_component=None,
                    )
                ],
            )
        return await self._generate_status_check(agent, user)

    # Handle get recent memories command (admin-only)
    if message.strip().lower() in [
        "/memories",
        "memories",
```

```
            "/recent_memories",
            "recent_memories",
        ]:
            # Check if user is admin
            if "admin" not in user.group_memberships:
                return WorkflowResult(
                    should_skip_llm=True,
                    components=[
                        UiComponent(
                            rich_component=RichTextComponent(
                                content="#   Access Denied\n\n"
                                "The `/memories` command is only available to administrators.\n\n"
                                "If you need access to memory management features, please contact your s
                                markdown=True,
                            ),
                            simple_component=None,
                        )
                    ],
                )
            return await self._get_recent_memories(agent, user, conversation)

        # Handle delete memory command (admin-only)
        if message.strip().lower().startswith("/delete "):
            # Check if user is admin
            if "admin" not in user.group_memberships:
                return WorkflowResult(
                    should_skip_llm=True,
                    components=[
                        UiComponent(
                            rich_component=RichTextComponent(
                                content="#   Access Denied\n\n"
                                "The `/delete` command is only available to administrators.\n\n"
                                "If you need access to memory management features, please contact your s
                                markdown=True,
                            ),
                            simple_component=None,
                        )
                    ],
                )
            memory_id = message.strip()[8:].strip()  # Extract ID after "/delete "
            return await self._delete_memory(agent, user, conversation, memory_id)

        # Don't handle other messages, pass to LLM
        return WorkflowResult(should_skip_llm=False)

    async def get_starter_ui(
        self, agent: "Agent", user: "User", conversation: "Conversation"
    ) -> Optional[List[UiComponent]]:
        """Generate starter UI based on available tools and setup status."""

        # Get available tools
        tools = await agent.tool_registry.get_schemas(user)
        tool_names = [tool.name for tool in tools]

        # Analyze setup
        setup_analysis = self._analyze_setup(tool_names)

        # Check if user is admin (has 'admin' in group memberships)
        is_admin = "admin" in user.group_memberships

        # Generate single concise card
        if self.welcome_message:
            # Use custom welcome message
            return [
                UiComponent(
                    rich_component=RichTextComponent(
                        content=self.welcome_message, markdown=True
                    ),
```

```
                    simple_component=None,
                )
            ]
        else:
            # Generate role-aware welcome card
            return [self._generate_starter_card(setup_analysis, is_admin)]

    def _generate_starter_card(
        self, analysis: Dict[str, Any], is_admin: bool
    ) -> UiComponent:
        """Generate a single concise starter card based on role and setup status."""

        if is_admin:
            # Admin view: includes setup status and memory management
            return self._generate_admin_starter_card(analysis)
        else:
            # User view: simple welcome message
            return self._generate_user_starter_card(analysis)

    def _generate_admin_starter_card(self, analysis: Dict[str, Any]) -> UiComponent:
        """Generate admin starter card with setup info and memory management."""

        # Build concise content
        if not analysis["has_sql"]:
            title = "Admin: Setup Required"
            content = "**  Admin View** - You have admin privileges and will see additional system infor
            status = "error"
            icon = "  "
        elif analysis["is_complete"]:
            title = "Admin: System Ready"
            content = "**  Admin View** - You have admin privileges and will see additional system infor
            content += "**Setup:** SQL   | Memory   | Visualization  "
            status = "success"
            icon = " "
```

Example: DefaultWorkflowHandler behaviors

The packaged example demonstrates how starter UI varies with tool availability.

```
"""
Example demonstrating the DefaultWorkflowHandler with setup health checking.

This example shows how the DefaultWorkflowHandler provides intelligent starter UI
that adapts based on available tools and helps users understand their setup status.

Run:
  PYTHONPATH=. python vanna/examples/default_workflow_handler_example.py
"""

import asyncio

from vanna import (
    AgentConfig,
    Agent,
    MemoryConversationStore,
    MockLlmService,
    User,
    DefaultWorkflowHandler,
)
from vanna.core.registry import ToolRegistry
from vanna.core.user.resolver import SimpleUserResolver
from vanna.tools import ListFilesTool


async def demonstrate_setup_scenarios():
    """Demonstrate different setup scenarios with DefaultWorkflowHandler."""
    print("  Starting DefaultWorkflowHandler Setup Health Check Demo\n")
```

```
# Create basic components
llm_service = MockLlmService(response_content="I'm ready to help!")
conversation_store = MemoryConversationStore()
user_resolver = SimpleUserResolver()

# Create test user
user = User(
    id="user1",
    username="alice",
    email="alice@example.com",
    group_memberships=["user"],
)

print("=" * 60)
print("SCENARIO 1: Empty Setup (No Tools)")
print("=" * 60)

# Empty tool registry
empty_registry = ToolRegistry()

agent_empty = Agent(
    llm_service=llm_service,
    tool_registry=empty_registry,
    user_resolver=user_resolver,
    conversation_store=conversation_store,
    config=AgentConfig(stream_responses=False),
    workflow_handler=DefaultWorkflowHandler(),
)

print("  Starter UI for empty setup:")
async for component in agent_empty.send_message(
    request_context=user_resolver.create_request_context(
        metadata={"starter_ui_request": True}
    ),
    message="",
    conversation_id="empty-setup",
):
    if hasattr(component, "simple_component") and component.simple_component:
        print(f"    {component.simple_component.text[:100]}...")
    elif hasattr(component, "rich_component"):
        comp = component.rich_component
        if hasattr(comp, "title"):
            print(f"    {comp.title}: {comp.status} - {comp.description}")
        elif hasattr(comp, "content"):
            print(f"    {comp.content[:100]}...")

print("\n" + "=" * 60)
print("SCENARIO 2: Functional Setup (SQL + Basic Tools)")
print("=" * 60)

# Tool registry with SQL tool (simulated)
functional_registry = ToolRegistry()

# Register a mock SQL tool (we'll simulate by tool name)
list_tool = ListFilesTool()
list_tool.name = "run_sql"  # Simulate SQL tool
functional_registry.register(list_tool)

agent_functional = Agent(
    llm_service=llm_service,
    tool_registry=functional_registry,
    user_resolver=user_resolver,
    conversation_store=conversation_store,
    config=AgentConfig(stream_responses=False),
    workflow_handler=DefaultWorkflowHandler(),
)

print("  Starter UI for functional setup:")
```

```
async for component in agent_functional.send_message(
    request_context=user_resolver.create_request_context(
        metadata={"starter_ui_request": True}
    ),
    message="",
    conversation_id="functional-setup",
):
    if hasattr(component, "simple_component") and component.simple_component:
        print(f"    {component.simple_component.text[:100]}...")
    elif hasattr(component, "rich_component"):
        comp = component.rich_component
        if hasattr(comp, "title"):
            print(f"    {comp.title}: {comp.status} - {comp.description}")
        elif hasattr(comp, "content"):
            print(f"    {comp.content[:100]}...")

print("\n" + "=" * 60)
print("SCENARIO 3: Complete Setup (SQL + Memory + Visualization)")
print("=" * 60)

# Complete tool registry
complete_registry = ToolRegistry()
```

8. Web UI

Vanna ships a ready-to-use web UI based on a Web Component (<vanna-chat>). The server templates include a demo page that mounts the component and points it at your API endpoints.

Demo HTML template excerpt (shows vanna-chat usage)

```
            Logged in as <span id="loggedInEmail" class="font-semibold text-vanna-navy"></span>
            <br>
            <button id="logoutButton" class="mt-2 px-3 py-1.5 bg-vanna-navy text-white text-xs rounded h
                Logout
            </button>
        </div>

        <!-- Chat Container (hidden by default) -->
        <div id="chatSections" class="hidden">
            <div class="bg-white rounded-xl shadow-lg h-[600px] overflow-hidden border border-vanna-teal
                <vanna-chat
                    api-base="{api_base_url}"
                    sse-endpoint="{api_base_url}/api/vanna/v2/chat_sse"
                    ws-endpoint="{api_base_url}/api/vanna/v2/chat_websocket"
                    poll-endpoint="{api_base_url}/api/vanna/v2/chat_poll">
                </vanna-chat>
            </div>

            <div class="mt-8 p-5 bg-white rounded-lg shadow border border-vanna-teal/30">
                <h3 class="text-lg font-semibold text-vanna-navy mb-3 font-serif">API Endpoints</h3>
                <ul class="space-y-2">
                    <li class="p-2 bg-vanna-cream/50 rounded font-mono text-sm">
                        <span class="font-bold text-vanna-teal mr-2">POST</span>{api_base_url}/api/vanna
                    </li>
                    <li class="p-2 bg-vanna-cream/50 rounded font-mono text-sm">
                        <span class="font-bold text-vanna-teal mr-2">WS</span>{api_base_url}/api/vanna/v
                    </li>
                    <li class="p-2 bg-vanna-cream/50 rounded font-mono text-sm">
                        <span class="font-bold text-vanna-teal mr-2">POST</span>{api_base_url}/api/vanna
                    </li>
                    <li class="p-2 bg-vanna-cream/50 rounded font-mono text-sm">
                        <span class="font-bold text-vanna-teal mr-2">GET</span>{api_base_url}/health - H
                    </li>
                </ul>
            </div>
        </div>
    </div>

    <script>
        // Cookie helpers
        const getCookie = (name) => {{
            const value = `; ${{document.cookie}}`;
            const parts = value.split(`; ${{name}}=`);
            return parts.length === 2 ? parts.pop().split(';').shift() : null;
        }};

        const setCookie = (name, value) => {{
            const expires = new Date(Date.now() + 365 * 864e5).toUTCString();
            document.cookie = `${{name}}=${{value}}; expires=${{expires}}; path=/; SameSite=Lax`;
        }};

        const deleteCookie = (name) => {{
            document.cookie = `${{name}}=; expires=Thu, 01 Jan 1970 00:00:00 UTC; path=/;`;
        }};

        // Login/Logout
        document.addEventListener('DOMContentLoaded', () => {{
```

```
            const email = getCookie('vanna_email');

            // Check if already logged in
```

Client server communication patterns

Streaming (SSE): the browser opens a server-sent events stream to receive chunks.

WebSockets: optional websocket endpoint for bidirectional streaming.

Polling: fallback mode for environments where streaming is constrained.

FastAPI routes (excerpt)

```
"""
FastAPI route implementations for Vanna Agents.
"""

import json
import traceback
from typing import Any, AsyncGenerator, Dict, Optional

from fastapi import FastAPI, HTTPException, Request, WebSocket, WebSocketDisconnect
from fastapi.responses import StreamingResponse, HTMLResponse

from ..base import ChatHandler, ChatRequest, ChatResponse
from ..base.templates import get_index_html
from ...core.user.request_context import RequestContext


def register_chat_routes(
    app: FastAPI, chat_handler: ChatHandler, config: Optional[Dict[str, Any]] = None
) -> None:
    """Register chat routes on FastAPI app.

    Args:
        app: FastAPI application
        chat_handler: Chat handler instance
        config: Server configuration
    """
    config = config or {}

    @app.get("/", response_class=HTMLResponse)
    async def index() -> str:
        """Serve the main chat interface."""
        dev_mode = config.get("dev_mode", False)
        cdn_url = config.get("cdn_url", "https://img.vanna.ai/vanna-components.js")
        api_base_url = config.get("api_base_url", "")

        return get_index_html(
            dev_mode=dev_mode, cdn_url=cdn_url, api_base_url=api_base_url
        )

    @app.post("/api/vanna/v2/chat_sse")
    async def chat_sse(
        chat_request: ChatRequest, http_request: Request
    ) -> StreamingResponse:
        """Server-Sent Events endpoint for streaming chat."""
        # Extract request context for user resolution
        chat_request.request_context = RequestContext(
            cookies=dict(http_request.cookies),
            headers=dict(http_request.headers),
            remote_addr=http_request.client.host if http_request.client else None,
            query_params=dict(http_request.query_params),
            metadata=chat_request.metadata,
        )
```

```
        async def generate() -> AsyncGenerator[str, None]:
            """Generate SSE stream."""
            try:
                async for chunk in chat_handler.handle_stream(chat_request):
                    chunk_json = chunk.model_dump_json()
                    yield f"data: {chunk_json}\n\n"
                yield "data: [DONE]\n\n"
            except Exception as e:
                traceback.print_stack()
                traceback.print_exc()
                error_data = {
                    "type": "error",
                    "data": {"message": str(e)},
                    "conversation_id": chat_request.conversation_id or "",
                    "request_id": chat_request.request_id or "",
                }
                yield f"data: {json.dumps(error_data)}\n\n"

        return StreamingResponse(
            generate(),
            media_type="text/event-stream",
            headers={
                "Cache-Control": "no-cache",
                "Connection": "keep-alive",
                "X-Accel-Buffering": "no",  # Disable nginx buffering
            },
        )

    @app.websocket("/api/vanna/v2/chat_websocket")
    async def chat_websocket(websocket: WebSocket) -> None:
        """WebSocket endpoint for real-time chat."""
        await websocket.accept()

        try:
            while True:
                # Receive message
                try:
                    data = await websocket.receive_json()

                    # Extract request context for user resolution
                    metadata = data.get("metadata", {})
                    data["request_context"] = RequestContext(
                        cookies=dict(websocket.cookies),
                        headers=dict(websocket.headers),
                        remote_addr=websocket.client.host if websocket.client else None,
                        query_params=dict(websocket.query_params),
                        metadata=metadata,
                    )

                    chat_request = ChatRequest(**data)
                except Exception as e:
                    traceback.print_stack()
                    traceback.print_exc()
                    await websocket.send_json(
                        {
                            "type": "error",
                            "data": {"message": f"Invalid request: {str(e)}"},
                        }
                    )
                    continue

                # Stream response
                try:
                    async for chunk in chat_handler.handle_stream(chat_request):
                        await websocket.send_json(chunk.model_dump())

                    # Send completion signal
                    await websocket.send_json(
```

```
                        {
                            "type": "completion",
                            "data": {"status": "done"},
                            "conversation_id": chunk.conversation_id
                            if "chunk" in locals()
                            else "",
                            "request_id": chunk.request_id
                            if "chunk" in locals()
                            else "",
                        }
                    )

                except Exception as e:
                    traceback.print_stack()
                    traceback.print_exc()
                    await websocket.send_json(
                        {
                            "type": "error",
                            "data": {"message": str(e)},
```

Flask routes (excerpt)

```
"""
Flask route implementations for Vanna Agents.
"""

import asyncio
import json
import traceback
from typing import Any, AsyncGenerator, Dict, Generator, Optional, Union

from flask import Flask, Response, jsonify, request

from ..base import ChatHandler, ChatRequest
from ..base.templates import get_index_html
from ...core.user.request_context import RequestContext


def register_chat_routes(
    app: Flask, chat_handler: ChatHandler, config: Optional[Dict[str, Any]] = None
) -> None:
    """Register chat routes on Flask app.

    Args:
        app: Flask application
        chat_handler: Chat handler instance
        config: Server configuration
    """
    config = config or {}

    @app.route("/")
    def index() -> str:
        """Serve the main chat interface."""
        dev_mode = config.get("dev_mode", False)
        cdn_url = config.get("cdn_url", "https://img.vanna.ai/vanna-components.js")
        api_base_url = config.get("api_base_url", "")

        return get_index_html(
            dev_mode=dev_mode, cdn_url=cdn_url, api_base_url=api_base_url
        )

    @app.route("/api/vanna/v2/chat_sse", methods=["POST"])
    def chat_sse() -> Union[Response, tuple[Response, int]]:
        """Server-Sent Events endpoint for streaming chat."""
        try:
            data = request.get_json()
            if not data:
                return jsonify({"error": "JSON body required"}), 400
```

```
            # Extract request context for user resolution
            data["request_context"] = RequestContext(
                cookies=dict(request.cookies),
                headers=dict(request.headers),
                remote_addr=request.remote_addr,
                query_params=dict(request.args),
            )

            chat_request = ChatRequest(**data)
        except Exception as e:
            traceback.print_stack()
            traceback.print_exc()
            return jsonify({"error": f"Invalid request: {str(e)}"}), 400

        def generate() -> Generator[str, None, None]:
            """Generate SSE stream."""
            loop = asyncio.new_event_loop()
            asyncio.set_event_loop(loop)

            try:

                async def async_generate() -> AsyncGenerator[str, None]:
                    async for chunk in chat_handler.handle_stream(chat_request):
                        chunk_json = chunk.model_dump_json()
                        yield f"data: {chunk_json}\n\n"

                gen = async_generate()
                try:
                    while True:
                        chunk = loop.run_until_complete(gen.__anext__())
                        yield chunk
                except StopAsyncIteration:
                    yield "data: [DONE]\n\n"
            finally:
                loop.close()

        return Response(
            generate(),
            mimetype="text/event-stream",
            headers={
                "Cache-Control": "no-cache",
                "Connection": "keep-alive",
                "X-Accel-Buffering": "no",  # Disable nginx buffering
            },
        )

    @app.route("/api/vanna/v2/chat_websocket")
    def chat_websocket() -> tuple[Response, int]:
        """WebSocket endpoint placeholder."""
        return jsonify(
            {
                "error": "WebSocket endpoint not implemented in basic Flask example",
                "suggestion": "Use Flask-SocketIO for WebSocket support",
            }
        ), 501

    @app.route("/api/vanna/v2/chat_poll", methods=["POST"])
    def chat_poll() -> Union[Response, tuple[Response, int]]:
        """Polling endpoint for chat."""
        try:
            data = request.get_json()
            if not data:
                return jsonify({"error": "JSON body required"}), 400

            # Extract request context for user resolution
            data["request_context"] = RequestContext(
                cookies=dict(request.cookies),
                headers=dict(request.headers),
```

```
            remote_addr=request.remote_addr,
            query_params=dict(request.args),
        )

        chat_request = ChatRequest(**data)
    except Exception as e:
        traceback.print_stack()
        traceback.print_exc()
        return jsonify({"error": f"Invalid request: {str(e)}"}), 400

    # Run async handler in new event loop
    loop = asyncio.new_event_loop()
    asyncio.set_event_loop(loop)
    try:
        result = loop.run_until_complete(chat_handler.handle_poll(chat_request))
        return jsonify(result.model_dump())
    except Exception as e:
        traceback.print_stack()
        traceback.print_exc()
        return jsonify({"error": f"Chat failed: {str(e)}"}), 500
    finally:
        loop.close()
```

Customization ideas

Point the component at a different API base URL (e.g., behind a reverse proxy).

Swap the HTML template with your own app shell and keep only the <vanna-chat> component + JS loader.

Implement your own auth mechanism and set cookies/headers that your UserResolver reads.

Add your own UI components (rich components) and render them in your UI client.

9. Deployment

You can deploy Vanna with either FastAPI or Flask. Both server implementations wrap an Agent and expose consistent HTTP endpoints for chat.

9.1 FastAPI deployment

Use VannaFastAPIServer to create and run a FastAPI app. It sets up CORS, static files (for the UI), and registers the chat routes.

VannaFastAPIServer excerpt

```
"""
FastAPI server factory for Vanna Agents.
"""

from typing import Any, Dict, Optional

from fastapi import FastAPI
from fastapi.middleware.cors import CORSMiddleware
from fastapi.staticfiles import StaticFiles

from ...core import Agent
from ..base import ChatHandler
from .routes import register_chat_routes


class VannaFastAPIServer:
    """FastAPI server factory for Vanna Agents."""

    def __init__(self, agent: Agent, config: Optional[Dict[str, Any]] = None):
        """Initialize FastAPI server.

        Args:
            agent: The agent to serve (must have user_resolver configured)
            config: Optional server configuration
        """
        self.agent = agent
        self.config = config or {}
        self.chat_handler = ChatHandler(agent)

    def create_app(self) -> FastAPI:
        """Create configured FastAPI app.

        Returns:
            Configured FastAPI application
        """
        # Create FastAPI app
        app_config = self.config.get("fastapi", {})
        app = FastAPI(
            title="Vanna Agents API",
            description="API server for Vanna Agents framework",
            version="0.1.0",
            **app_config,
        )

        # Configure CORS if enabled
        cors_config = self.config.get("cors", {})
        if cors_config.get("enabled", True):
            cors_params = {k: v for k, v in cors_config.items() if k != "enabled"}

            # Set sensible defaults
            cors_params.setdefault("allow_origins", ["*"])
            cors_params.setdefault("allow_credentials", True)
```

```
            cors_params.setdefault("allow_methods", ["*"])
            cors_params.setdefault("allow_headers", ["*"])

            app.add_middleware(CORSMiddleware, **cors_params)

        # Add static file serving in dev mode
        dev_mode = self.config.get("dev_mode", False)
        if dev_mode:
            static_folder = self.config.get("static_folder", "static")
            try:
                import os

                if os.path.exists(static_folder):
                    app.mount(
                        "/static", StaticFiles(directory=static_folder), name="static"
                    )
            except Exception:
                pass  # Static files not available

        # Register routes
        register_chat_routes(app, self.chat_handler, self.config)

        # Add health check
        @app.get("/health")
        async def health_check() -> Dict[str, str]:
            return {"status": "healthy", "service": "vanna"}

        return app

    def run(self, **kwargs: Any) -> None:
        """Run the FastAPI server.

        This method automatically detects if running in an async environment
        (Jupyter, Colab, IPython, etc.) and:
        - Uses appropriate async handling for existing event loops
        - Sets up port forwarding if in Google Colab
        - Displays the correct URL for accessing the app

        Args:
            **kwargs: Arguments passed to uvicorn configuration
        """
        import sys
        import asyncio
        import uvicorn

        # Check if we're in an environment with a running event loop FIRST
        in_async_env = False
        try:
            asyncio.get_running_loop()
            in_async_env = True
        except RuntimeError:
            in_async_env = False

        # If in async environment, apply nest_asyncio BEFORE creating the app
        if in_async_env:
            try:
                import nest_asyncio

                nest_asyncio.apply()
            except ImportError:
                print("Warning: nest_asyncio not installed. Installing...")
                import subprocess

                subprocess.check_call(
                    [sys.executable, "-m", "pip", "install", "nest_asyncio"]
                )
                import nest_asyncio
```

```
                nest_asyncio.apply()

        # Now create the app after nest_asyncio is applied
        app = self.create_app()

        # Set defaults
        run_kwargs = {"host": "0.0.0.0", "port": 8000, "log_level": "info", **kwargs}

        # Get the port and other config from run_kwargs
        port = run_kwargs.get("port", 8000)
        host = run_kwargs.get("host", "0.0.0.0")
        log_level = run_kwargs.get("log_level", "info")

        # Check if we're specifically in Google Colab for port forwarding
        in_colab = "google.colab" in sys.modules

        if in_colab:
            try:
                from google.colab import output

                output.serve_kernel_port_as_window(port)
                from google.colab.output import eval_js

                print("Your app is running at:")
                print(eval_js(f"google.colab.kernel.proxyPort({port})"))
            except Exception as e:
                print(f"Warning: Could not set up Colab port forwarding: {e}")
                print(f"Your app is running at: http://localhost:{port}")
        else:
            print("Your app is running at:")
            print(f"http://localhost:{port}")

        if in_async_env:
            # In Jupyter/Colab, create config with loop="asyncio" and use asyncio.run()
            # This matches the working pattern from Colab
            config = uvicorn.Config(
                app, host=host, port=port, log_level=log_level, loop="asyncio"
            )
            server = uvicorn.Server(config)
            asyncio.run(server.serve())
        else:
            # Normal execution outside of Jupyter/Colab
            uvicorn.run(app, **run_kwargs)
```

Typical usage

```
from vanna import Agent
from vanna.servers.fastapi import VannaFastAPIServer

agent = ...  # build your agent
server = VannaFastAPIServer(agent=agent, dev_mode=True)
server.run(host="0.0.0.0", port=8000)  # uses uvicorn
```

Production notes

Put FastAPI behind a reverse proxy (NGINX/Traefik) and terminate TLS there.

Set CORS carefully (restrict origins) if you embed the UI on a different domain.

Prefer streaming endpoints for best UX; ensure your proxy supports SSE/WebSockets.

9.2 Flask deployment

VannaFlaskServer excerpt

```
"""
Flask server factory for Vanna Agents.
```

```
"""

import asyncio
from typing import Any, Dict, Optional

from flask import Flask
from flask_cors import CORS

from ...core import Agent
from ..base import ChatHandler
from .routes import register_chat_routes


class VannaFlaskServer:
    """Flask server factory for Vanna Agents."""

    def __init__(self, agent: Agent, config: Optional[Dict[str, Any]] = None):
        """Initialize Flask server.

        Args:
            agent: The agent to serve (must have user_resolver configured)
            config: Optional server configuration
        """
        self.agent = agent
        self.config = config or {}
        self.chat_handler = ChatHandler(agent)

    def create_app(self) -> Flask:
        """Create configured Flask app.

        Returns:
            Configured Flask application
        """
        # Check if dev mode is enabled
        dev_mode = self.config.get("dev_mode", False)
        static_folder = self.config.get("static_folder", "static") if dev_mode else None

        app = Flask(__name__, static_folder=static_folder, static_url_path="/static")

        # Apply configuration
        app.config.update(self.config.get("flask", {}))

        # Enable CORS if configured
        cors_config = self.config.get("cors", {})
        if cors_config.get("enabled", True):
            CORS(app, **{k: v for k, v in cors_config.items() if k != "enabled"})

        # Register routes
        register_chat_routes(app, self.chat_handler, self.config)

        # Add health check
        @app.route("/health")
        def health_check() -> Dict[str, str]:
            return {"status": "healthy", "service": "vanna"}

        return app

    def run(self, **kwargs: Any) -> None:
        """Run the Flask server.

        This method automatically detects if running in an async environment
        (Jupyter, Colab, IPython, etc.) and:
        - Installs and applies nest_asyncio to handle existing event loops
        - Sets up port forwarding if in Google Colab
        - Displays the correct URL for accessing the app

        Args:
            **kwargs: Arguments passed to Flask.run()
```

```
        """
        import sys

        app = self.create_app()

        # Set defaults
        run_kwargs = {"host": "0.0.0.0", "port": 5000, "debug": False, **kwargs}

        # Get the port from run_kwargs
        port = run_kwargs.get("port", 5000)

        # Check if we're in an environment with a running event loop
        # (Jupyter, Colab, IPython, VS Code notebooks, etc.)
        in_async_env = False
        try:
            import asyncio

            try:
                asyncio.get_running_loop()
                in_async_env = True
            except RuntimeError:
                in_async_env = False
        except Exception:
            pass

        if in_async_env:
            # Apply nest_asyncio to allow nested event loops
            try:
                import nest_asyncio

                nest_asyncio.apply()
            except ImportError:
                print("Warning: nest_asyncio not installed. Installing...")
                import subprocess

                subprocess.check_call(
                    [sys.executable, "-m", "pip", "install", "nest_asyncio"]
                )
                import nest_asyncio

                nest_asyncio.apply()

        # Check if we're specifically in Google Colab for port forwarding
        in_colab = "google.colab" in sys.modules

        if in_colab:
            try:
                from google.colab import output

                output.serve_kernel_port_as_window(port)
                from google.colab.output import eval_js

                print("Your app is running at:")
                print(eval_js(f"google.colab.kernel.proxyPort({port})"))
            except Exception as e:
                print(f"Warning: Could not set up Colab port forwarding: {e}")
                print(f"Your app is running at: http://localhost:{port}")
        else:
            print("Your app is running at:")
            print(f"http://localhost:{port}")

        app.run(**run_kwargs)
```

Typical usage

```
from vanna.servers.flask import VannaFlaskServer

agent = ...  # build your agent
```

```
server = VannaFlaskServer(agent=agent, dev_mode=True)
server.run(host="0.0.0.0", port=8080)
```

Deployment checklist

Configure secrets via environment variables or a secret manager (don t hardcode API keys).

Add authentication/authorization (see the Auth section) before exposing publicly.

Enable logging, observability and audit logging in production.

Set resource limits: maximum SQL runtime, maximum rows, and timeouts for LLM calls.

10. Authentication and Authorization

Authentication in Vanna is modeled as resolving a user for each request. The server layer supplies a RequestContext (cookies, headers, remote address). Your UserResolver turns that into a User.

User model (excerpt)

```
"""
User domain models.

This module contains data models for user management.
"""

from typing import Any, Dict, List, Optional

from pydantic import BaseModel, ConfigDict, Field


class User(BaseModel):
    """User model for authentication and scoping."""

    id: str = Field(description="Unique user identifier")
    username: Optional[str] = Field(default=None, description="Username")
    email: Optional[str] = Field(default=None, description="User email")
    metadata: Dict[str, Any] = Field(
        default_factory=dict, description="Additional user metadata"
    )
    group_memberships: List[str] = Field(
        default_factory=list, description="Groups the user belongs to"
    )

    model_config = ConfigDict(extra="allow")
```

UserResolver implementations (excerpt)

```
"""
User resolver interface for web request authentication.

This module provides the abstract base class for resolving web requests
to authenticated User objects.
"""

from abc import ABC, abstractmethod

from .models import User
from .request_context import RequestContext


class UserResolver(ABC):
    """Resolves web requests to authenticated users.

    Implementations of this interface handle the specifics of extracting
    user identity from request context (cookies, headers, tokens, etc.)
    and creating authenticated User objects.

    Example:
        class JwtUserResolver(UserResolver):
            async def resolve_user(self, request_context: RequestContext) -> User:
                token = request_context.get_header('Authorization')
                # ... validate JWT and extract user info
                return User(id=user_id, username=username, email=email)
    """
```

```
    @abstractmethod
    async def resolve_user(self, request_context: RequestContext) -> User:
        """Resolve user from request context.

        Args:
            request_context: Structured request context with cookies, headers, etc.

        Returns:
            Authenticated User object

        Raises:
            Can raise exceptions for authentication failures
        """
        pass
```

Cookie-based authentication example

The packaged mock_auth_example demonstrates extracting the user identity from a cookie and injecting it via middleware.

```
"""
Mock authentication example to verify user resolution is working.

This example demonstrates the new UserResolver architecture where:
1. UserResolver is a required parameter of Agent
2. Agent.send_message() accepts RequestContext (not User directly)
3. The Agent resolves the user internally using the UserResolver

The agent uses an LLM middleware to inject user info into the response,
so we can verify the authentication is working correctly.

Usage:
    python -m vanna.examples.mock_auth_example
"""

from __future__ import annotations

import asyncio

from vanna import AgentConfig, Agent
from vanna.core.registry import ToolRegistry
from vanna.core.llm import LlmRequest, LlmResponse
from vanna.core.middleware import LlmMiddleware
from vanna.integrations.mock import MockLlmService
from vanna.core.user import CookieEmailUserResolver, RequestContext


class UserEchoMiddleware(LlmMiddleware):
    """Middleware that injects user email into LLM responses."""

    async def after_llm_response(
        self, request: LlmRequest, response: LlmResponse
    ) -> LlmResponse:
        """Inject user email into response."""
        # Extract user email from request user_id (which is set to user.id in the agent)
        user_id = request.user_id

        # Create a new response with user info
        new_content = f"Hello! You are authenticated as: {user_id}"

        return LlmResponse(
            content=new_content,
            finish_reason=response.finish_reason,
            usage=response.usage,
        )

def create_demo_agent() -> Agent:
```

```
    """Create a demo agent for server usage.

    Returns:
        Configured Agent instance with cookie-based authentication
    """
    # Create a mock LLM
    llm_service = MockLlmService(response_content="Mock response")

    # Empty tool registry
    tool_registry = ToolRegistry()

    # Cookie-based user resolver
    user_resolver = CookieEmailUserResolver(cookie_name="vanna_email")

    # User echo middleware
    middleware = UserEchoMiddleware()

    # Create agent with user resolver and middleware
    agent = Agent(
        llm_service=llm_service,
        tool_registry=tool_registry,
        user_resolver=user_resolver,
        llm_middlewares=[middleware],
        config=AgentConfig(
            stream_responses=True,
            include_thinking_indicators=False,
        ),
    )

    return agent


async def demo_authentication():
    """Demonstrate authentication with different request contexts."""
    agent = create_demo_agent()

    print("=== Mock Authentication Demo ===")
    print("This example verifies that user resolution is working correctly.\n")

    # Test 1: Request with email cookie
    print("  Test 1: Authenticated user (alice@example.com)")
    request_context = RequestContext(
        cookies={"vanna_email": "alice@example.com"},
        headers={},
        remote_addr="127.0.0.1",
    )

    print(
        "Request context:",
        {
            "cookies": request_context.cookies,
            "headers": request_context.headers,
            "remote_addr": request_context.remote_addr,
        },
    )

    # Send message - Agent will resolve user internally
    agent_response = ""
    async for component in agent.send_message(
        request_context=request_context,
        message="Who am I?",
        conversation_id="test_conv_1",
    ):
        # Extract and display user info from the resolved user
        if hasattr(component, "rich_component"):
            rich = component.rich_component
            # Check if it's a text component
            if rich.type.value == "text":
```

```
                # Access content directly from the component (before serialization)
                if hasattr(rich, "content"):
                    agent_response = rich.content
```

Authorization: tool access control

Tools can be restricted by group membership. The Tool base and ToolRegistry enforce this before execution.

```
"""
Tool domain interface.

This module contains the abstract base class for tools.
"""

from abc import ABC, abstractmethod
from typing import Generic, List, Type, TypeVar

from .models import ToolContext, ToolResult, ToolSchema

# Type variable for tool argument types
T = TypeVar("T")


class Tool(ABC, Generic[T]):
    """Abstract base class for tools."""

    @property
    @abstractmethod
    def name(self) -> str:
        """Unique name for this tool."""
        pass

    @property
    @abstractmethod
    def description(self) -> str:
        """Description of what this tool does."""
        pass

    @property
    def access_groups(self) -> List[str]:
        """Groups permitted to access this tool."""
        return []

    @abstractmethod
    def get_args_schema(self) -> Type[T]:
        """Return the Pydantic model for arguments."""
        pass

    @abstractmethod
    async def execute(self, context: ToolContext, args: T) -> ToolResult:
        """Execute the tool with validated arguments.

        Args:
            context: Execution context containing user, conversation_id, and request_id
            args: Validated tool arguments

        Returns:
            ToolResult with success status, result for LLM, and optional UI component
        """
        pass

    def get_schema(self) -> ToolSchema:
        """Generate tool schema for LLM."""
        from typing import Any, cast

        args_model = self.get_args_schema()
```

```
        # Get the schema - args_model should be a Pydantic model class
        schema = (
            cast(Any, args_model).model_json_schema()
            if hasattr(args_model, "model_json_schema")
            else {}
        )
        return ToolSchema(
            name=self.name,
            description=self.description,
            parameters=schema,
            access_groups=self.access_groups,
        )
```

Email-based authentication pattern (tool-driven)

If you want the agent to explicitly authenticate a user during chat, you can add a dedicated authentication tool. The packaged example includes an AuthTool.

```
    async def has_permission(self, user: User, permission: str) -> bool:
        """Check if user has permission."""
        return permission in user.permissions

    def _is_valid_email(self, email: str) -> bool:
        """Simple email validation."""
        return "@" in email and "." in email.split("@")[1]


# Authentication Tool
class AuthArgs(BaseModel):
    """Arguments for authentication."""

    email: str = Field(description="User's email address")


class AuthTool(Tool[AuthArgs]):
    """Tool to authenticate users by email."""

    def __init__(self, user_service: DemoEmailUserService):
        self.user_service = user_service

    @property
    def name(self) -> str:
        return "authenticate_user"

    @property
    def description(self) -> str:
        return "Authenticate a user by their email address. Use this when the user provides an email."

    def get_args_schema(self) -> Type[AuthArgs]:
        return AuthArgs

    async def execute(self, context: ToolContext, args: AuthArgs) -> ToolResult:
        """Execute authentication."""
        user = await self.user_service.authenticate({"email": args.email})

        if user:
            success_msg = (
                f"  Welcome {user.username}! You're now authenticated as {user.email}"
            )

            auth_component = RichComponent(
                type="status_card",
                data={
                    "title": "Authentication Success",
                    "status": "success",
                    "description": success_msg,
                    "icon": " ",
```

```
                    "metadata": {
                        "user_id": user.id,
                        "username": user.username,
                        "email": user.email,
                    },
                },
            )

            return ToolResult(
                success=True,
                result_for_llm=f"User successfully authenticated as {user.username} ({user.email}). They
                ui_component=UiComponent(rich_component=auth_component),
            )
        else:
            error_msg = f"  Invalid email format: {args.email}"
            error_component = RichComponent(
                type="status_card",
                data={
                    "title": "Authentication Failed",
                    "status": "error",
                    "description": error_msg,
                    "icon": " ",
                    "metadata": {"email": args.email},
                },
            )

            return ToolResult(
                success=False,
                result_for_llm=f"Authentication failed for {args.email}. Please provide a valid email ad
                ui_component=UiComponent(rich_component=error_component),
                error=error_msg,
            )


def create_demo_agent() -> Agent:
    """Create a demo agent for REPL and server usage.

    Returns:
        Configured Agent instance with email authentication
    """
    return create_auth_agent()


def create_auth_agent() -> Agent:
    """Create agent with email authentication."""

    # Create user service
    user_service = DemoEmailUserService()

    # Use simple mock LLM - the system prompt will guide behavior
    llm_service = MockLlmService(
        response_content="Hello! I'm your AI assistant. To provide you with personalized help, I'll need
    )

    # Create tool registry with auth tool
    tool_registry = ToolRegistry()
    auth_tool = AuthTool(user_service)
    tool_registry.register(auth_tool)

    # Create agent with authentication system prompt
    agent = Agent(
        llm_service=llm_service,
        config=AgentConfig(
            stream_responses=True,
            include_thinking_indicators=False,  # Cleaner output for demo
            system_prompt="""You are a helpful AI assistant with an email-based authentication system.

AUTHENTICATION BEHAVIOR:
```

```
1. When a user provides an email address in their message, immediately use the 'authenticate_user' tool
2. Look for emails in patterns like "my email is...", "I'm john@example.com", or any text with @ symbols
3. If user isn't authenticated, politely ask for their email address to get started
```

Security recommendations

Prefer server-side authentication (cookies/JWTs) over user tells the agent their email for real deployments.

Never trust LLM output for authorization decisions; enforce permissions in code.

If your SQL tool can touch sensitive data, enforce database-level least privilege (read-only roles, row/column filters).

Log authentication events (audit logging) and consider rate limiting.

11. UI Features, Charts, and Artifacts

Vanna streams UiComponent objects from the agent to the server. The server turns those into JSON chunks that the web UI renders in real time.

Component taxonomy

```
"""UI Component system for Vanna Agents."""

# Base component
from .base import UiComponent

# Simple components
from .simple import (
    SimpleComponent,
    SimpleComponentType,
    SimpleTextComponent,
    SimpleImageComponent,
    SimpleLinkComponent,
)

# Rich components - re-export all
from .rich import (
    # Base
    RichComponent,
    ComponentType,
    ComponentLifecycle,
    # Text
    RichTextComponent,
    # Data
    DataFrameComponent,
    ChartComponent,
    # Feedback
    NotificationComponent,
    StatusCardComponent,
    ProgressBarComponent,
    ProgressDisplayComponent,
    StatusIndicatorComponent,
    LogViewerComponent,
    LogEntry,
    BadgeComponent,
    IconTextComponent,
    # Interactive
    TaskListComponent,
    Task,
    StatusBarUpdateComponent,
    TaskTrackerUpdateComponent,
    ChatInputUpdateComponent,
    TaskOperation,
    ButtonComponent,
    ButtonGroupComponent,
    # Containers
    CardComponent,
    # Specialized
    ArtifactComponent,
)

__all__ = [
    # Base
    "UiComponent",
    # Simple components
    "SimpleComponent",
    "SimpleComponentType",
    "SimpleTextComponent",
    "SimpleImageComponent",
```

```
    "SimpleLinkComponent",
    # Rich components - Base
    "RichComponent",
    "ComponentType",
    "ComponentLifecycle",
    # Rich components - Text
    "RichTextComponent",
    # Rich components - Data
    "DataFrameComponent",
    "ChartComponent",
    # Rich components - Feedback
    "NotificationComponent",
    "StatusCardComponent",
    "ProgressBarComponent",
    "ProgressDisplayComponent",
    "StatusIndicatorComponent",
    "LogViewerComponent",
    "LogEntry",
    "BadgeComponent",
    "IconTextComponent",
    # Rich components - Interactive
    "TaskListComponent",
    "Task",
    "StatusBarUpdateComponent",
    "TaskTrackerUpdateComponent",
    "ChatInputUpdateComponent",
    "TaskOperation",
    "ButtonComponent",
    "ButtonGroupComponent",
    # Rich components - Containers
    "CardComponent",
    # Rich components - Specialized
    "ArtifactComponent",
]
```

Charts

There are two common ways to show charts:

Use VisualizeDataTool to create a Plotly chart from a DataFrame.

Use an ArtifactComponent to emit HTML/D3 widgets that your host UI can open in a separate panel.

VisualizeDataTool excerpt

```
"""Tool for visualizing DataFrame data from CSV files."""

from typing import Optional, Type
import logging
import pandas as pd
from pydantic import BaseModel, Field

from vanna.core.tool import Tool, ToolContext, ToolResult
from vanna.components import (
    UiComponent,
    ChartComponent,
    NotificationComponent,
    ComponentType,
    SimpleTextComponent,
)

from .file_system import FileSystem, LocalFileSystem
from vanna.integrations.plotly import PlotlyChartGenerator

logger = logging.getLogger(__name__)
```

```
class VisualizeDataArgs(BaseModel):
    """Arguments for visualize_data tool."""

    filename: str = Field(description="Name of the CSV file to visualize")
    title: Optional[str] = Field(
        default=None, description="Optional title for the chart"
    )


class VisualizeDataTool(Tool[VisualizeDataArgs]):
    """Tool that reads CSV files and generates visualizations using dependency injection."""

    def __init__(
        self,
        file_system: Optional[FileSystem] = None,
        plotly_generator: Optional[PlotlyChartGenerator] = None,
    ):
        """Initialize the tool with FileSystem and PlotlyChartGenerator.

        Args:
            file_system: FileSystem implementation for reading CSV files (defaults to LocalFileSystem)
            plotly_generator: PlotlyChartGenerator for creating Plotly charts (defaults to PlotlyChartGe
        """
        self.file_system = file_system or LocalFileSystem()
        self.plotly_generator = plotly_generator or PlotlyChartGenerator()

    @property
    def name(self) -> str:
        return "visualize_data"

    @property
    def description(self) -> str:
        return "Create a visualization from a CSV file. The tool automatically selects an appropriate ch

    def get_args_schema(self) -> Type[VisualizeDataArgs]:
        return VisualizeDataArgs

    async def execute(
        self, context: ToolContext, args: VisualizeDataArgs
    ) -> ToolResult:
        """Read CSV file and generate visualization."""
        try:
            logger.info(f"Starting visualization for file: {args.filename}")

            # Read the CSV file using FileSystem
            csv_content = await self.file_system.read_file(args.filename, context)
            logger.info(f"Read {len(csv_content)} bytes from CSV file")

            # Parse CSV into DataFrame
            import io

            df = pd.read_csv(io.StringIO(csv_content))
            logger.info(
                f"Parsed DataFrame with shape {df.shape}, columns: {df.columns.tolist()}, dtypes: {df.dt
            )

            # Generate title
            title = args.title or f"Visualization of {args.filename}"

            # Generate chart using PlotlyChartGenerator
            logger.info("Generating chart...")
            chart_dict = self.plotly_generator.generate_chart(df, title)
            logger.info(
                f"Chart generated, type: {type(chart_dict)}, keys: {list(chart_dict.keys()) if isinstanc
            )

            # Create result message
            row_count = len(df)
```

```
            col_count = len(df.columns)
            result = f"Created visualization from '{args.filename}' ({row_count} rows, {col_count} colum

            # Create ChartComponent
            logger.info("Creating ChartComponent...")
            chart_component = ChartComponent(
                chart_type="plotly",
                data=chart_dict,
                title=title,
                config={
                    "data_shape": {"rows": row_count, "columns": col_count},
                    "source_file": args.filename,
                },
            )
            logger.info("ChartComponent created successfully")

            logger.info("Creating ToolResult...")
            tool_result = ToolResult(
                success=True,
                result_for_llm=result,
                ui_component=UiComponent(
                    rich_component=chart_component,
                    simple_component=SimpleTextComponent(text=result),
                ),
                metadata={
                    "filename": args.filename,
                    "rows": row_count,
                    "columns": col_count,
                    "chart": chart_dict,
                },
            )
            logger.info("ToolResult created successfully")
            return tool_result

        except FileNotFoundError as e:
            logger.error(f"File not found: {args.filename}", exc_info=True)
            error_message = f"File not found: {args.filename}"
            return ToolResult(
                success=False,
                result_for_llm=error_message,
                ui_component=UiComponent(
                    rich_component=NotificationComponent(
                        type=ComponentType.NOTIFICATION,
                        level="error",
                        message=error_message,
                    ),
                    simple_component=SimpleTextComponent(text=error_message),
                ),
                error=str(e),
                metadata={"error_type": "file_not_found"},
            )
        except pd.errors.ParserError as e:
            logger.error(f"CSV parse error for {args.filename}", exc_info=True)
            error_message = f"Failed to parse CSV file '{args.filename}': {str(e)}"
            return ToolResult(
                success=False,
                result_for_llm=error_message,
                ui_component=UiComponent(
                    rich_component=NotificationComponent(
                        type=ComponentType.NOTIFICATION,
                        level="error",
                        message=error_message,
                    ),
                    simple_component=SimpleTextComponent(text=error_message),
                ),
                error=str(e),
                metadata={"error_type": "csv_parse_error"},
            )
```

```
        except ValueError as e:
            logger.error(f"Visualization error for {args.filename}", exc_info=True)
            error_message = f"Cannot visualize data: {str(e)}"
            return ToolResult(
                success=False,
                result_for_llm=error_message,
                ui_component=UiComponent(
                    rich_component=NotificationComponent(
                        type=ComponentType.NOTIFICATION,
                        level="error",
                        message=error_message,
                    ),
                    simple_component=SimpleTextComponent(text=error_message),
                ),
                error=str(e),
                metadata={"error_type": "visualization_error"},
            )
        except Exception as e:
            logger.error(
                f"Unexpected error creating visualization for {args.filename}",
                exc_info=True,
            )
            error_message = f"Error creating visualization: {str(e)}"
            return ToolResult(
                success=False,
                result_for_llm=error_message,
                ui_component=UiComponent(
                    rich_component=NotificationComponent(
                        type=ComponentType.NOTIFICATION,
                        level="error",
                        message=error_message,
                    ),
                    simple_component=SimpleTextComponent(text=error_message),
                ),
                error=str(e),
                metadata={"error_type": "general_error"},
            )
```

Plotly chart generator (excerpt)

```
"""Plotly-based chart generator with automatic chart type selection."""

from typing import Dict, Any, List, cast
import json
import pandas as pd
import plotly.graph_objects as go
import plotly.express as px
import plotly.io as pio


class PlotlyChartGenerator:
    """Generate Plotly charts using heuristics based on DataFrame characteristics."""

    # Vanna brand colors from landing page
    THEME_COLORS = {
        "navy": "#023d60",
        "cream": "#e7e1cf",
        "teal": "#15a8a8",
        "orange": "#fe5d26",
        "magenta": "#bf1363",
    }

    # Color palette for charts (excluding cream as it's too light for data)
    COLOR_PALETTE = ["#15a8a8", "#fe5d26", "#bf1363", "#023d60"]

    def generate_chart(self, df: pd.DataFrame, title: str = "Chart") -> Dict[str, Any]:
        """Generate a Plotly chart based on DataFrame shape and types.
```

```
        Heuristics:
        - 4+ columns: table
        - 1 numeric column: histogram
        - 2 columns (1 categorical, 1 numeric): bar chart
        - 2 numeric columns: scatter plot
        - 3+ numeric columns: correlation heatmap or multi-line chart
        - Time series data: line chart
        - Multiple categorical: grouped bar chart

        Args:
            df: DataFrame to visualize
            title: Title for the chart

        Returns:
            Plotly figure as dictionary

        Raises:
            ValueError: If DataFrame is empty or cannot be visualized
        """
        if df.empty:
            raise ValueError("Cannot visualize empty DataFrame")

        # Heuristic: If 4 or more columns, render as a table
        if len(df.columns) >= 4:
            fig = self._create_table(df, title)
            result: Dict[str, Any] = json.loads(pio.to_json(fig))
            return result

        # Identify column types
        numeric_cols = df.select_dtypes(include=["number"]).columns.tolist()
        categorical_cols = df.select_dtypes(
            include=["object", "category"]
        ).columns.tolist()
        datetime_cols = df.select_dtypes(include=["datetime64"]).columns.tolist()

        # Check for time series
        is_timeseries = len(datetime_cols) > 0

        # Apply heuristics
        if is_timeseries and len(numeric_cols) > 0:
            # Time series line chart
            fig = self._create_time_series_chart(
                df, datetime_cols[0], numeric_cols, title
            )
        elif len(numeric_cols) == 1 and len(categorical_cols) == 0:
            # Single numeric column: histogram
            fig = self._create_histogram(df, numeric_cols[0], title)
        elif len(numeric_cols) == 1 and len(categorical_cols) == 1:
            # One categorical, one numeric: bar chart
            fig = self._create_bar_chart(
                df, categorical_cols[0], numeric_cols[0], title
            )
        elif len(numeric_cols) == 2:
            # Two numeric columns: scatter plot
            fig = self._create_scatter_plot(df, numeric_cols[0], numeric_cols[1], title)
        elif len(numeric_cols) >= 3:
            # Multiple numeric columns: correlation heatmap
            fig = self._create_correlation_heatmap(df, numeric_cols, title)
        elif len(categorical_cols) >= 2:
            # Multiple categorical: grouped bar chart
            fig = self._create_grouped_bar_chart(df, categorical_cols, title)
        else:
            # Fallback: show first two columns as scatter/bar
            if len(df.columns) >= 2:
                fig = self._create_generic_chart(
                    df, df.columns[0], df.columns[1], title
                )
            else:
```

```
                raise ValueError(
                    "Cannot determine appropriate visualization for this DataFrame"
                )

        # Convert to JSON-serializable dict using plotly's JSON encoder
        result = json.loads(pio.to_json(fig))
        return result

    def _apply_standard_layout(self, fig: go.Figure) -> go.Figure:
        """Apply consistent Vanna brand styling to all charts.

        Uses Vanna brand colors from the landing page for a cohesive look.

        Args:
            fig: Plotly figure to update

        Returns:
            Updated figure with Vanna brand styling
        """
        fig.update_layout(
            # paper_bgcolor='white',
            # plot_bgcolor='white',
            font={"color": self.THEME_COLORS["navy"]},  # Navy for text
            autosize=True,  # Allow chart to resize responsively
            colorway=self.COLOR_PALETTE,  # Use Vanna brand colors for data
            # Don't set width/height - let frontend handle sizing
        )
        return fig

    def _create_histogram(self, df: pd.DataFrame, column: str, title: str) -> go.Figure:
        """Create a histogram for a single numeric column."""
        fig = px.histogram(
            df,
            x=column,
            title=title,
            color_discrete_sequence=[self.THEME_COLORS["teal"]],
        )
        fig.update_layout(xaxis_title=column, yaxis_title="Count", showlegend=False)
        self._apply_standard_layout(fig)
        return fig

    def _create_bar_chart(
        self, df: pd.DataFrame, x_col: str, y_col: str, title: str
    ) -> go.Figure:
```

Artifacts (interactive external renderers)

ArtifactComponent allows the agent to emit an openable artifact that your UI can render externally (e.g., in a modal or side panel).

Artifact example excerpt

```
#!/usr/bin/env python3
"""
Example demonstrating the artifact system in Vanna Agents.

This script shows how agents can create interactive artifacts that can be
rendered externally by developers listening for the 'artifact-opened' event.
"""


import asyncio
from typing import AsyncGenerator, Optional

from vanna import Agent, UiComponent, User, AgentConfig
from vanna.core.rich_components import ArtifactComponent
from vanna.integrations.anthropic.mock import MockLlmService
```

```
from vanna.core.interfaces import Agent, LlmService


class ArtifactDemoAgent(Agent):
    """Demo agent that creates various types of artifacts."""

    def __init__(self, llm_service: Optional[LlmService] = None) -> None:
        if llm_service is None:
            llm_service = MockLlmService(
                "I'll help you create interactive artifacts! Try asking me to create a chart, dashboard,
            )
        super().__init__(
            llm_service=llm_service,
            config=AgentConfig(
                stream_responses=True,
                include_thinking_indicators=True,
            ),
        )

    async def send_message(
        self, user: User, message: str, *, conversation_id: Optional[str] = None
    ) -> AsyncGenerator[UiComponent, None]:
        """Handle user messages and create appropriate artifacts."""
        # First send the normal response
        async for component in super().send_message(
            user, message, conversation_id=conversation_id
        ):
            yield component

        # Then create artifacts based on message content
        message_lower = message.lower()

        if any(
            word in message_lower for word in ["chart", "graph", "visualization", "d3"]
        ):
            async for component in self.create_d3_visualization():
                yield component
        elif any(
            word in message_lower for word in ["dashboard", "analytics", "metrics"]
        ):
            async for component in self.create_dashboard_artifact():
                yield component
        elif any(
            word in message_lower for word in ["html", "interactive", "widget", "demo"]
        ):
            async for component in self.create_html_artifact():
                yield component

    async def create_html_artifact(self) -> AsyncGenerator[UiComponent, None]:
        """Create a simple HTML artifact."""
        html_content = """
        <div style="padding: 20px; font-family: Arial, sans-serif;">
            <h2 style="color: #333;">Interactive HTML Artifact</h2>
            <p>This is a simple HTML artifact that can be opened externally.</p>
            <button onclick="alert('Hello from the artifact!')">Click me!</button>
            <div style="margin-top: 20px;">
                <input type="text" placeholder="Type something..." id="textInput">
                <button onclick="document.getElementById('output').textContent = document.getElementById
                    Update Text
                </button>
            </div>
            <div id="output" style="margin-top: 10px; padding: 10px; background: #f0f0f0; border-radius:
                Output will appear here...
            </div>
        </div>
        """

        artifact = ArtifactComponent.create_html(
```

```
            content=html_content,
            title="Interactive HTML Demo",
            description="A simple HTML artifact with interactive elements",
        )

        yield UiComponent(rich_component=artifact)

    async def create_d3_visualization(self) -> AsyncGenerator[UiComponent, None]:
        """Create a D3.js visualization artifact."""
        d3_content = """
        <div id="chart" style="width: 100%; height: 400px;"></div>
        <script>
            // Sample data
            const data = [
                {name: 'A', value: 30},
                {name: 'B', value: 80},
                {name: 'C', value: 45},
                {name: 'D', value: 60},
                {name: 'E', value: 20},
                {name: 'F', value: 90}
            ];

            // Set up dimensions
            const margin = {top: 20, right: 30, bottom: 40, left: 40};
            const width = 800 - margin.left - margin.right;
            const height = 400 - margin.top - margin.bottom;

            // Create SVG
            const svg = d3.select("#chart")
                .append("svg")
                .attr("width", width + margin.left + margin.right)
                .attr("height", height + margin.top + margin.bottom)
                .append("g")
                .attr("transform", `translate(${margin.left},${margin.top})`);

            // Create scales
            const xScale = d3.scaleBand()
                .domain(data.map(d => d.name))
```

Rich UI demos in the examples

primitive_components_demo.py demonstrates basic rich components.

mock_rich_components_demo.py shows streaming updates for status/progress/task lists.

visualization_example.py uses visualization pathways.

12. System Prompts and Context Enhancers

The system prompt guides the LLM s behavior. Vanna supports:

Static system prompt via AgentConfig.system_prompt

Dynamic system prompt via a SystemPromptBuilder implementation

Optional LLM context enhancers that can inject additional context (e.g., memory results) into the prompt or messages

Default system prompt builder (excerpt)

```
"""
Default system prompt builder implementation with memory workflow support.

This module provides a default implementation of the SystemPromptBuilder interface
that automatically includes memory workflow instructions when memory tools are available.
"""

from typing import TYPE_CHECKING, List, Optional
from datetime import datetime

from .base import SystemPromptBuilder

if TYPE_CHECKING:
    from ..tool.models import ToolSchema
    from ..user.models import User


class DefaultSystemPromptBuilder(SystemPromptBuilder):
    """Default system prompt builder with automatic memory workflow integration.

    Dynamically generates system prompts that include memory workflow
    instructions when memory tools (search_saved_correct_tool_uses and
    save_question_tool_args) are available.
    """

    def __init__(self, base_prompt: Optional[str] = None):
        """Initialize with an optional base prompt.

        Args:
            base_prompt: Optional base system prompt. If not provided, uses a default.
        """
        self.base_prompt = base_prompt

    async def build_system_prompt(
        self, user: "User", tools: List["ToolSchema"]
    ) -> Optional[str]:
        """
        Build a system prompt with memory workflow instructions.

        Args:
            user: The user making the request
            tools: List of tools available to the user

        Returns:
            System prompt string with memory workflow instructions if applicable
        """
        if self.base_prompt is not None:
            return self.base_prompt

        # Check which memory tools are available
```

```
        tool_names = [tool.name for tool in tools]
        has_search = "search_saved_correct_tool_uses" in tool_names
        has_save = "save_question_tool_args" in tool_names
        has_text_memory = "save_text_memory" in tool_names

        # Get today's date
        today_date = datetime.now().strftime("%Y-%m-%d")

        # Base system prompt
        prompt_parts = [
            f"You are Vanna, an AI data analyst assistant created to help users with data analysis tasks
            "",
            "Response Guidelines:",
            "- Any summary of what you did or observations should be the final step.",
            "- Use the available tools to help the user accomplish their goals.",
            "- When you execute a query, that raw result is shown to the user outside of your response s
        ]

        if tools:
            prompt_parts.append(
                f"\nYou have access to the following tools: {', '.join(tool_names)}"
            )

        # Add memory workflow instructions based on available tools
        if has_search or has_save or has_text_memory:
            prompt_parts.append("\n" + "=" * 60)
            prompt_parts.append("MEMORY SYSTEM:")
            prompt_parts.append("=" * 60)

        if has_search or has_save:
            prompt_parts.append("\n1. TOOL USAGE MEMORY (Structured Workflow):")
            prompt_parts.append("-" * 50)

        if has_search:
            prompt_parts.extend(
                [
                    "",
                    "  BEFORE executing any tool (run_sql, visualize_data, or calculator), you MUST firs
                    "",
                    "  Review the search results (if any) to inform your approach before proceeding with
                ]
            )

        if has_save:
            prompt_parts.extend(
                [
                    "",
                    "  AFTER successfully executing a tool that produces correct and useful results, you
                ]
            )

        if has_search or has_save:
            prompt_parts.extend(
                [
                    "",
                    "Example workflow:",
                    "    User asks a question",
                    f'    First: Call search_saved_correct_tool_uses(question="user\'s question")'
                    if has_search
                    else "",
                    "    Then: Execute the appropriate tool(s) based on search results and the question"
                    f'    Finally: If successful, call save_question_tool_args(question="user\'s questio
                    if has_save
                    else "",
                    "",
                    "Do NOT skip the search step, even if you think you know how to answer. Do NOT forge
                    if has_search
                    else "",
```

```
                    "",
                    "The only exceptions to searching first are:",
                    '    When the user is explicitly asking about the tools themselves (like "list the t
                    "    When the user is testing or asking you to demonstrate the save/search functiona
                ]
            )

        if has_text_memory:
            prompt_parts.extend(
                [
                    "",
                    "2. TEXT MEMORY (Domain Knowledge & Context):",
                    "-" * 50,
                    "",
                    "  save_text_memory: Save important context about the database, schema, or domain",
                    "",
                    "Use text memory to save:",
                    "    Database schema details (column meanings, data types, relationships)",
                    "    Company-specific terminology and definitions",
                    "    Query patterns or best practices for this database",
                    "    Domain knowledge about the business or data",
                    "    User preferences for queries or visualizations",
                    "",
                    "DO NOT save:",
                    "    Information already captured in tool usage memory",
                    "    One-time query results or temporary observations",
                    "",
                    "Examples:",
                    '    save_text_memory(content="The status column uses 1 for active, 0 for inactive")',
                    '    save_text_memory(content="MRR means Monthly Recurring Revenue in our schema")',
                    "    save_text_memory(content=\"Always exclude test accounts where email contains 't
                ]
            )

        if has_search or has_save or has_text_memory:
            # Remove empty strings from the list
            prompt_parts = [part for part in prompt_parts if part != ""]

        return "\n".join(prompt_parts)
```

LLM context enhancers

Enhancers can add extra context (e.g., relevant memories, schema hints, user metadata). The default enhancer composes enrichers into a final context block.

```
"""
Default LLM context enhancer implementation using AgentMemory.

This implementation enriches the system prompt with relevant memories
based on the user's initial message.
"""

from typing import TYPE_CHECKING, List, Optional
from .base import LlmContextEnhancer

if TYPE_CHECKING:
    from ..user.models import User
    from ..llm.models import LlmMessage
    from ...capabilities.agent_memory import AgentMemory, TextMemorySearchResult


class DefaultLlmContextEnhancer(LlmContextEnhancer):
    """Default enhancer that uses AgentMemory to add relevant context.

    This enhancer searches the agent's memory for relevant examples and
    tool use patterns based on the user's message, and adds them to the
    system prompt.
```

```
    Example:
        agent = Agent(
            llm_service=...,
            agent_memory=agent_memory,
            llm_context_enhancer=DefaultLlmContextEnhancer(agent_memory)
        )
    """

    def __init__(self, agent_memory: Optional["AgentMemory"] = None):
        """Initialize with optional agent memory.

        Args:
            agent_memory: Optional AgentMemory instance. If not provided,
                          enhancement will be skipped.
        """
        self.agent_memory = agent_memory

    async def enhance_system_prompt(
        self, system_prompt: str, user_message: str, user: "User"
    ) -> str:
        """Enhance system prompt with relevant memories.

        Searches agent memory for relevant text memories based on the
        user's message and adds them to the system prompt.

        Args:
            system_prompt: The original system prompt
            user_message: The initial user message
            user: The user making the request

        Returns:
            Enhanced system prompt with relevant examples from memory
        """
        if not self.agent_memory:
            return system_prompt

        try:
            # Import here to avoid circular dependency
            from ..tool import ToolContext
            import uuid

            # Create a temporary context for memory search
            context = ToolContext(
                user=user,
                conversation_id="temp",
                request_id=str(uuid.uuid4()),
                agent_memory=self.agent_memory,
            )

            # Search for relevant text memories based on user message
            memories: List[
                "TextMemorySearchResult"
            ] = await self.agent_memory.search_text_memories(
                query=user_message, context=context, limit=5
            )

            if not memories:
                return system_prompt

            # Format memories as context snippets to add to system prompt
            examples_section = "\n\n## Relevant Context from Memory\n\n"
            examples_section += "The following domain knowledge and context from prior interactions may

            for result in memories:
                memory = result.memory
                examples_section += f"  {memory.content}\n"

            # Append examples to system prompt
```

```
            return system_prompt + examples_section

        except Exception as e:
            # If memory search fails, return original prompt
            # Don't fail the entire request due to memory issues
            import logging

            logger = logging.getLogger(__name__)
            logger.warning(f"Failed to enhance system prompt with memories: {e}")
            return system_prompt

    async def enhance_user_messages(
        self, messages: list["LlmMessage"], user: "User"
    ) -> list["LlmMessage"]:
        """Enhance user messages.

        The default implementation doesn't modify user messages.
        Override this to add context to user messages if needed.

        Args:
            messages: The list of messages
            user: The user making the request

        Returns:
            Original list of messages (unmodified)
        """
        return messages
```

Context enrichers

```
"""
Base context enricher interface.

Context enrichers allow you to add additional data to the ToolContext
before tools are executed.
"""

from abc import ABC
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from ..tool.models import ToolContext


class ToolContextEnricher(ABC):
    """Enricher for adding data to ToolContext.

    Subclass this to create custom enrichers that can:
    - Add user preferences from database
    - Inject session state
    - Add temporal context (timezone, current date)
    - Include user history or profile data
    - Add environment-specific configuration

    Example:
        class UserPreferencesEnricher(ToolContextEnricher):
            def __init__(self, db):
                self.db = db

            async def enrich_context(self, context: ToolContext) -> ToolContext:
                # Fetch user preferences
                prefs = await self.db.get_user_preferences(context.user.id)

                # Add to context metadata
                context.metadata["preferences"] = prefs
                context.metadata["timezone"] = prefs.get("timezone", "UTC")
```

```
                return context

        agent = AgentRunner(
            llm_service=...,
            context_enrichers=[UserPreferencesEnricher(db), SessionEnricher()]
        )
    """

    async def enrich_context(self, context: "ToolContext") -> "ToolContext":
        """Enrich the tool execution context with additional data.

        Args:
            context: The tool context to enrich

        Returns:
            Enriched context (typically modified in-place)

        Note:
            Enrichers typically modify the context.metadata dict to add
            additional data that tools can access.
        """
        return context
```

Custom system prompt examples

The packaged example shows how to build prompts based on the user and the available tools:

```
"""
Example demonstrating custom system prompt builder with dependency injection.

This example shows how to create a custom SystemPromptBuilder that dynamically
generates system prompts based on user context and available tools.

Usage:
  python -m vanna.examples.custom_system_prompt_example
"""

from typing import List, Optional

from vanna.core.interfaces import SystemPromptBuilder
from vanna.core.models import ToolSchema, User


class CustomSystemPromptBuilder(SystemPromptBuilder):
    """Custom system prompt builder that personalizes prompts based on user."""

    async def build_system_prompt(
        self, user: User, tools: List[ToolSchema]
    ) -> Optional[str]:
        """Build a personalized system prompt.

        Args:
            user: The user making the request
            tools: List of tools available to the user

        Returns:
            Personalized system prompt
        """
        # Build personalized greeting
        username = user.username or user.id
        greeting = f"Hello {username}! I'm your AI assistant."

        # Add role-specific instructions based on user permissions
        role_instructions = []
        if "admin" in user.permissions:
            role_instructions.append(
```

```
                "As an admin user, you have access to all tools and capabilities."
            )
        elif "analyst" in user.permissions:
            role_instructions.append(
                "You're working as an analyst. I'll help you query and visualize data."
            )
        else:
            role_instructions.append("I'm here to help you with your tasks.")

        # List available tools
        tool_info = []
        if tools:
            tool_info.append("\nAvailable tools:")
            for tool in tools:
                tool_info.append(f"- {tool.name}: {tool.description}")

        # Combine all parts
        parts = [greeting] + role_instructions + tool_info
        return "\n".join(parts)


class SQLAssistantSystemPromptBuilder(SystemPromptBuilder):
    """System prompt builder specifically for SQL database assistants."""

    def __init__(self, database_name: str = "database"):
        """Initialize with database context.

        Args:
            database_name: Name of the database being queried
        """
        self.database_name = database_name

    async def build_system_prompt(
        self, user: User, tools: List[ToolSchema]
    ) -> Optional[str]:
        """Build a SQL-focused system prompt.

        Args:
            user: The user making the request
            tools: List of tools available to the user

        Returns:
            SQL-focused system prompt
        """
        prompt = f"""You are an expert SQL database assistant for the {self.database_name} database.

Your primary responsibilities:
1. Write efficient, correct SQL queries
2. Explain query results clearly
3. Suggest optimizations when relevant
4. Visualize data when appropriate

Guidelines:
- Always validate SQL syntax before execution
- Use appropriate JOINs and avoid Cartesian products
- Limit result sets to reasonable sizes by default
- Format numbers and dates for readability
"""

        # Add tool-specific instructions
        has_viz_tool = any(tool.name == "visualize_data" for tool in tools)
        if has_viz_tool:
            prompt += "\n- Create visualizations for numerical data when it helps understanding"

        return prompt


async def demo() -> None:
```

```
    """Demonstrate custom system prompt builders."""
    from vanna import Agent, User
    from vanna.core.registry import ToolRegistry
```

13. Lifecycle Hooks and LLM Middlewares

Two complementary extension points:

Lifecycle hooks: run before/after message handling at the agent level (good for quotas, logging, moderation, preprocessing).

LLM middlewares: wrap the LLM request/response (good for adding headers, injecting user info, rewriting prompts, logging tokens, or measuring latency).

LifecycleHook interface (excerpt)

```
"""
Base lifecycle hook interface.

Lifecycle hooks allow you to intercept and customize agent behavior
at key points in the execution flow.
"""

from abc import ABC
from typing import TYPE_CHECKING, Any, Optional

if TYPE_CHECKING:
    from ..user.models import User
    from ..tool import Tool
    from ..tool.models import ToolContext, ToolResult


class LifecycleHook(ABC):
    """Hook into agent execution lifecycle.

    Subclass this to create custom hooks that can:
    - Modify messages before processing
    - Add logging or telemetry
    - Enforce quotas or rate limits
    - Transform tool results
    - Add custom validation

    Example:
        class LoggingHook(LifecycleHook):
            async def before_message(self, user: User, message: str) -> Optional[str]:
                print(f"User {user.username} sent: {message}")
                return None  # Don't modify

        agent = AgentRunner(
            llm_service=...,
            lifecycle_hooks=[LoggingHook(), QuotaCheckHook()]
        )
    """

    async def before_message(self, user: "User", message: str) -> Optional[str]:
        """Called before processing a user message.

        Args:
            user: User sending the message
            message: Original message content

        Returns:
            Modified message string, or None to keep original

        Raises:
            AgentError: To halt message processing (e.g., quota exceeded)
        """
        return None
```

```
    async def after_message(self, result: Any) -> None:
        """Called after message has been fully processed.

        Args:
            result: Final result from message processing
        """
        pass

    async def before_tool(self, tool: "Tool[Any]", context: "ToolContext") -> None:
        """Called before tool execution.

        Args:
            tool: Tool about to be executed
            context: Tool execution context

        Raises:
            AgentError: To prevent tool execution
        """
        pass

    async def after_tool(self, result: "ToolResult") -> Optional["ToolResult"]:
        """Called after tool execution.

        Args:
            result: Result from tool execution

        Returns:
            Modified ToolResult, or None to keep original
        """
        return None
```

LlmMiddleware interface (excerpt)

```
"""
Base LLM middleware interface.

Middleware allows you to intercept and transform LLM requests and responses
for caching, monitoring, content filtering, and more.
"""

from abc import ABC
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    from ..llm import LlmRequest, LlmResponse


class LlmMiddleware(ABC):
    """Middleware for intercepting LLM requests and responses.

    Subclass this to create custom middleware that can:
    - Cache LLM responses
    - Log requests/responses
    - Filter or modify content
    - Track costs and usage
    - Implement fallback strategies

    Example:
        class CachingMiddleware(LlmMiddleware):
            def __init__(self):
                self.cache = {}

            async def before_llm_request(self, request: LlmRequest) -> LlmRequest:
                # Could check cache here
                return request

            async def after_llm_response(self, request: LlmRequest, response: LlmResponse) -> LlmRespons
```

```
                # Cache the response
                cache_key = self._compute_key(request)
                self.cache[cache_key] = response
                return response

        agent = AgentRunner(
            llm_service=...,
            llm_middlewares=[CachingMiddleware(), LoggingMiddleware()]
        )
    """

    async def before_llm_request(self, request: "LlmRequest") -> "LlmRequest":
        """Called before sending request to LLM.

        Args:
            request: The LLM request about to be sent

        Returns:
            Modified request, or original if no changes
        """
        return request

    async def after_llm_response(
        self, request: "LlmRequest", response: "LlmResponse"
    ) -> "LlmResponse":
        """Called after receiving response from LLM.

        Args:
            request: The original request
            response: The LLM response

        Returns:
            Modified response, or original if no changes
        """
        return response
```

Example: Quota enforcement using lifecycle hooks

```
"""
Example demonstrating lifecycle hooks for user quota management.

This example shows how to use lifecycle hooks to add custom functionality
like quota management without creating custom agent runner subclasses.
"""

from typing import Any, Dict, Optional
from vanna.core import Agent, LifecycleHook, User
from vanna.core.errors import AgentError


class QuotaExceededError(AgentError):
    """Raised when a user exceeds their message quota."""

    pass


class QuotaCheckHook(LifecycleHook):
    """Lifecycle hook that enforces user-based message quotas."""

    def __init__(self, default_quota: int = 10) -> None:
        """Initialize quota hook.

        Args:
            default_quota: Default quota per user if not specifically set
        """
        self._user_quotas: Dict[str, int] = {}
        self._user_usage: Dict[str, int] = {}
```

```
        self._default_quota = default_quota

    def set_user_quota(self, user_id: str, quota: int) -> None:
        """Set a specific quota for a user."""
        self._user_quotas[user_id] = quota

    def get_user_quota(self, user_id: str) -> int:
        """Get the quota for a user."""
        return self._user_quotas.get(user_id, self._default_quota)

    def get_user_usage(self, user_id: str) -> int:
        """Get current usage count for a user."""
        return self._user_usage.get(user_id, 0)

    def get_user_remaining(self, user_id: str) -> int:
        """Get remaining messages for a user."""
        return self.get_user_quota(user_id) - self.get_user_usage(user_id)

    def reset_user_usage(self, user_id: str) -> None:
        """Reset usage count for a user."""
        self._user_usage[user_id] = 0

    async def before_message(self, user: User, message: str) -> Optional[str]:
        """Check quota before processing message.

        Raises:
            QuotaExceededError: If user has exceeded their quota
        """
        usage = self.get_user_usage(user.id)
        quota = self.get_user_quota(user.id)

        if usage >= quota:
            raise QuotaExceededError(
                f"User {user.username} has exceeded their quota of {quota} messages. "
                f"Current usage: {usage}"
            )

        # Increment usage count
        current_usage = self._user_usage.get(user.id, 0)
        self._user_usage[user.id] = current_usage + 1

        # Don't modify the message
        return None


class LoggingHook(LifecycleHook):
    """Example logging hook for demonstration."""

    async def before_message(self, user: User, message: str) -> Optional[str]:
        """Log incoming messages."""
        print(f"[LOG] User {user.username} ({user.id}) sent message: {message[:50]}...")
        return None

    async def after_message(self, result: Any) -> None:
        """Log message completion."""
        print(f"[LOG] Message processing completed")


async def run_example() -> None:
    """
    Example showing how to use lifecycle hooks with Agent.

    Instead of creating a custom subclass, we compose
    the behavior using lifecycle hooks.
    """
    from vanna.core.registry import ToolRegistry
    from vanna.integrations.anthropic import AnthropicLlmService
    from vanna.integrations.local import MemoryConversationStore
```

```
    # Create quota hook
    quota_hook = QuotaCheckHook(default_quota=10)
    quota_hook.set_user_quota("user123", 5)  # Set custom quota for specific user

    # Create logging hook
    logging_hook = LoggingHook()

    # Create agent with multiple hooks
    agent = Agent(
        llm_service=AnthropicLlmService(api_key="your-api-key"),
        tool_registry=ToolRegistry(),
        conversation_store=MemoryConversationStore(),
        lifecycle_hooks=[
            logging_hook,  # Logs will happen first
            quota_hook,  # Then quota check
        ],
    )

    # Create a test user
    user = User(
        id="user123", username="test_user", email="test@example.com", permissions=[]
    )

    # Send messages - will track quota
    try:
        async for component in agent.send_message(user=user, message="Hello, agent!"):
            # Process UI components
            pass

        # Check remaining quota
        remaining = quota_hook.get_user_remaining(user.id)
        print(f"Remaining messages: {remaining}/{quota_hook.get_user_quota(user.id)}")
```

Example: injecting user info via middleware

```
from __future__ import annotations

import asyncio

from vanna import AgentConfig, Agent
from vanna.core.registry import ToolRegistry
from vanna.core.llm import LlmRequest, LlmResponse
from vanna.core.middleware import LlmMiddleware
from vanna.integrations.mock import MockLlmService
from vanna.core.user import CookieEmailUserResolver, RequestContext


class UserEchoMiddleware(LlmMiddleware):
    """Middleware that injects user email into LLM responses."""

    async def after_llm_response(
        self, request: LlmRequest, response: LlmResponse
    ) -> LlmResponse:
        """Inject user email into response."""
        # Extract user email from request user_id (which is set to user.id in the agent)
        user_id = request.user_id

        # Create a new response with user info
        new_content = f"Hello! You are authenticated as: {user_id}"

        return LlmResponse(
            content=new_content,
            finish_reason=response.finish_reason,
            usage=response.usage,
        )

def create_demo_agent() -> Agent:
```

```
    """Create a demo agent for server usage.

    Returns:
        Configured Agent instance with cookie-based authentication
    """
    # Create a mock LLM
    llm_service = MockLlmService(response_content="Mock response")

    # Empty tool registry
    tool_registry = ToolRegistry()

    # Cookie-based user resolver
    user_resolver = CookieEmailUserResolver(cookie_name="vanna_email")

    # User echo middleware
    middleware = UserEchoMiddleware()

    # Create agent with user resolver and middleware
    agent = Agent(
        llm_service=llm_service,
        tool_registry=tool_registry,
        user_resolver=user_resolver,
```

14. Error Recovery and Conversation Filters

Agents in production need robust handling for:

LLM timeouts and transient API failures

Tool execution failures (DB errors, validation errors, permission errors)

User input errors or ambiguous requests

Streaming interruptions

Recovery strategy interface (excerpt)

```
"""
Base error recovery strategy interface.

Recovery strategies allow you to customize how the agent handles errors
during tool execution and LLM communication.
"""

from abc import ABC
from typing import TYPE_CHECKING

from .models import RecoveryAction, RecoveryActionType

if TYPE_CHECKING:
    from ..tool.models import ToolContext
    from ..llm import LlmRequest


class ErrorRecoveryStrategy(ABC):
    """Strategy for handling errors and implementing retry logic.

    Subclass this to create custom error recovery strategies that can:
    - Retry failed operations with backoff
    - Fallback to alternative approaches
    - Log errors to external systems
    - Gracefully degrade functionality

    Example:
        class ExponentialBackoffStrategy(ErrorRecoveryStrategy):
            async def handle_tool_error(
                self, error: Exception, context: ToolContext, attempt: int
            ) -> RecoveryAction:
                if attempt < 3:
                    delay = (2 ** attempt) * 1000  # Exponential backoff
                    return RecoveryAction(
                        action=RecoveryActionType.RETRY,
                        retry_delay_ms=delay,
                        message=f"Retrying after {delay}ms"
                    )
                return RecoveryAction(
                    action=RecoveryActionType.FAIL,
                    message="Max retries exceeded"
                )

        agent = AgentRunner(
            llm_service=...,
            error_recovery_strategy=ExponentialBackoffStrategy()
        )
    """

    async def handle_tool_error(
```

```
        self, error: Exception, context: "ToolContext", attempt: int = 1
    ) -> RecoveryAction:
        """Handle errors during tool execution.

        Args:
            error: The exception that occurred
            context: Tool execution context
            attempt: Current attempt number (1-indexed)

        Returns:
            RecoveryAction indicating how to proceed
        """
        # Default: fail immediately
        return RecoveryAction(
            action=RecoveryActionType.FAIL, message=f"Tool error: {str(error)}"
        )

    async def handle_llm_error(
        self, error: Exception, request: "LlmRequest", attempt: int = 1
    ) -> RecoveryAction:
        """Handle errors during LLM communication.

        Args:
            error: The exception that occurred
            request: The LLM request that failed
            attempt: Current attempt number (1-indexed)

        Returns:
            RecoveryAction indicating how to proceed
        """
        # Default: fail immediately
        return RecoveryAction(
            action=RecoveryActionType.FAIL, message=f"LLM error: {str(error)}"
        )
```

Recovery models (excerpt)

```
"""
Recovery action models for error handling.
"""

from enum import Enum
from typing import Any, Optional

from pydantic import BaseModel, Field


class RecoveryActionType(str, Enum):
    """Types of recovery actions."""

    RETRY = "retry"
    FAIL = "fail"
    FALLBACK = "fallback"
    SKIP = "skip"


class RecoveryAction(BaseModel):
    """Action to take when recovering from an error."""

    action: RecoveryActionType = Field(description="Type of recovery action")
    retry_delay_ms: Optional[int] = Field(
        default=None, description="Delay before retry in milliseconds"
    )
    fallback_value: Optional[Any] = Field(
        default=None, description="Fallback value to use"
    )
    message: Optional[str] = Field(
        default=None, description="Message to include with action"
```

```
    )
```

Conversation filters (excerpt)

Filters can modify the messages or block messages before they reach the LLM (e.g., remove sensitive info, enforce policies).

```
"""
Base conversation filter interface.

Conversation filters allow you to transform conversation history before
it's sent to the LLM for processing.
"""

from abc import ABC
from typing import TYPE_CHECKING, List

if TYPE_CHECKING:
    from ..storage import Message


class ConversationFilter(ABC):
    """Filter for transforming conversation history.

    Subclass this to create custom filters that can:
    - Remove sensitive information
    - Summarize long conversations
    - Manage context window limits
    - Deduplicate similar messages
    - Prioritize recent or relevant messages

    Example:
        class ContextWindowFilter(ConversationFilter):
            def __init__(self, max_tokens: int = 8000):
                self.max_tokens = max_tokens

            async def filter_messages(self, messages: List[Message]) -> List[Message]:
                # Estimate tokens (rough approximation)
                total_tokens = 0
                filtered = []

                # Keep system message and recent messages
                for msg in reversed(messages):
                    msg_tokens = len(msg.content or "") // 4
                    if total_tokens + msg_tokens > self.max_tokens:
                        break
                    filtered.insert(0, msg)
                    total_tokens += msg_tokens

                return filtered

        agent = AgentRunner(
            llm_service=...,
            conversation_filters=[
                SensitiveDataFilter(),
                ContextWindowFilter(max_tokens=8000)
            ]
        )
    """

    async def filter_messages(self, messages: List["Message"]) -> List["Message"]:
        """Filter and transform conversation messages.

        Args:
            messages: List of conversation messages

        Returns:
```

```
            Filtered/transformed list of messages

        Note:
            Filters are applied in order, so messages passed to later
            filters may already be modified by earlier filters.
        """
        return messages
```

Recommended patterns

For LLM calls: exponential backoff retries on 429/5xx (with a max cap).

For SQL: show user-friendly error messages (syntax, permissions, timeouts), and keep raw error details out of the UI unless needed.

For dangerous tools (file system / python): constrain to a sandbox directory and disallow network access.

15. Observability and Audit Logging

Observability and audit logging are separate concerns:

Observability: performance/health telemetry (spans, metrics, traces) to monitor the system.

Audit logging: security/forensics log of sensitive actions (tool executions, authentication, data access).

ObservabilityProvider interface (excerpt)

```
"""
Base observability provider interface.

Observability providers allow you to collect telemetry data about
agent execution for monitoring and debugging.
"""

from abc import ABC
from typing import Any, Dict, Optional

from .models import Span, Metric


class ObservabilityProvider(ABC):
    """Provider for collecting telemetry and observability data.

    Subclass this to create custom observability integrations that can:
    - Emit metrics to monitoring systems
    - Create distributed traces
    - Log performance data
    - Track costs and usage
    - Monitor error rates

    Example:
        class PrometheusProvider(ObservabilityProvider):
            def __init__(self, registry):
                self.registry = registry
                self.request_counter = Counter(
                    'agent_requests_total',
                    'Total agent requests',
                    registry=registry
                )

            async def record_metric(self, name: str, value: float, tags: Dict[str, str]) -> None:
                if name == "agent.request":
                    self.request_counter.inc()

            async def create_span(self, name: str, attributes: Optional[Dict[str, Any]] = None) -> Span:
                span = Span(name=name, attributes=attributes or {})
                return span

        agent = AgentRunner(
            llm_service=...,
            observability_provider=PrometheusProvider(registry)
        )
    """

    async def record_metric(
        self,
        name: str,
        value: float,
        unit: str = "",
        tags: Optional[Dict[str, str]] = None,
```

```
    ) -> None:
        """Record a metric measurement.

        Args:
            name: Metric name (e.g., "agent.request.duration")
            value: Metric value
            unit: Unit of measurement (e.g., "ms", "tokens")
            tags: Additional tags/labels for the metric
        """
        pass

    async def create_span(
        self, name: str, attributes: Optional[Dict[str, Any]] = None
    ) -> Span:
        """Create a new span for tracing.

        Args:
            name: Span name/operation
            attributes: Initial span attributes

        Returns:
            Span object to track the operation

        Note:
            Call span.end() when the operation completes.
        """
        return Span(name=name, attributes=attributes or {})

    async def end_span(self, span: Span) -> None:
        """End a span and record it.

        Args:
            span: The span to end
        """
        span.end()
```

AuditLogger interface + event models (excerpt)

```
"""
Base audit logger interface.

Audit loggers enable tracking user actions, tool invocations, and access control
decisions for security, compliance, and debugging.
"""

import hashlib
from abc import ABC, abstractmethod
from datetime import datetime
from typing import TYPE_CHECKING, Any, Dict, List, Optional

from .models import (
    AiResponseEvent,
    AuditEvent,
    ToolAccessCheckEvent,
    ToolInvocationEvent,
    ToolResultEvent,
    UiFeatureAccessCheckEvent,
)

if TYPE_CHECKING:
    from ..tool.models import ToolCall, ToolContext, ToolResult
    from ..user.models import User


class AuditLogger(ABC):
    """Abstract base class for audit logging implementations.
```

```
    Implementations can:
    - Write to files (JSON, CSV, etc.)
    - Send to databases (Postgres, MongoDB, etc.)
    - Stream to cloud services (CloudWatch, Datadog, etc.)
    - Send to SIEM systems (Splunk, Elastic, etc.)

    Example:
        class PostgresAuditLogger(AuditLogger):
            async def log_event(self, event: AuditEvent) -> None:
                await self.db.execute(
                    "INSERT INTO audit_log (...) VALUES (...)",
                    event.model_dump()
                )

        agent = Agent(
            llm_service=...,
            audit_logger=PostgresAuditLogger(db_pool)
        )
    """

    @abstractmethod
    async def log_event(self, event: AuditEvent) -> None:
        """Log a single audit event.

        Args:
            event: The audit event to log

        Raises:
            Exception: If logging fails critically
        """
        pass

    async def log_tool_access_check(
        self,
        user: "User",
        tool_name: str,
        access_granted: bool,
        required_groups: List[str],
        context: "ToolContext",
        reason: Optional[str] = None,
    ) -> None:
        """Convenience method for logging tool access checks.

        Args:
            user: User attempting to access the tool
            tool_name: Name of the tool being accessed
            access_granted: Whether access was granted
            required_groups: Groups required to access the tool
            context: Tool execution context
            reason: Optional reason for denial
        """
        event = ToolAccessCheckEvent(
            user_id=user.id,
            username=user.username,
            user_email=user.email,
            user_groups=user.group_memberships,
            conversation_id=context.conversation_id,
            request_id=context.request_id,
            tool_name=tool_name,
            access_granted=access_granted,
            required_groups=required_groups,
            reason=reason,
        )
        await self.log_event(event)

    async def log_tool_invocation(
        self,
        user: "User",
```

```
        tool_call: "ToolCall",
        ui_features: List[str],
        context: "ToolContext",
        sanitize_parameters: bool = True,
    ) -> None:
        """Convenience method for logging tool invocations.

        Args:
            user: User invoking the tool
            tool_call: Tool call information
            ui_features: List of UI features available to the user
            context: Tool execution context
            sanitize_parameters: Whether to sanitize sensitive parameters
        """
        parameters = tool_call.arguments.copy()
        sanitized = False

        if sanitize_parameters:
            parameters, sanitized = self._sanitize_parameters(parameters)

        event = ToolInvocationEvent(
            user_id=user.id,
            username=user.username,
            user_email=user.email,
            user_groups=user.group_memberships,
            conversation_id=context.conversation_id,
            request_id=context.request_id,
            tool_call_id=tool_call.id,
            tool_name=tool_call.name,
            parameters=parameters,
            parameters_sanitized=sanitized,
            ui_features_available=ui_features,
        )
        await self.log_event(event)

    async def log_tool_result(
        self,
        user: "User",
        tool_call: "ToolCall",
        result: "ToolResult",
        context: "ToolContext",
    ) -> None:
        """Convenience method for logging tool results.

        Args:
            user: User who invoked the tool
            tool_call: Tool call information
            result: Tool execution result
            context: Tool execution context
        """
        event = ToolResultEvent(
            user_id=user.id,
            username=user.username,
            user_email=user.email,
            user_groups=user.group_memberships,
            conversation_id=context.conversation_id,
            request_id=context.request_id,
            tool_call_id=tool_call.id,
            tool_name=tool_call.name,
            success=result.success,
            error=result.error,
            execution_time_ms=result.metadata.get("execution_time_ms", 0.0),
            result_size_bytes=(
                len(result.result_for_llm.encode("utf-8"))
                if result.result_for_llm
                else 0
            ),
            ui_component_type=(
```

```
                result.ui_component.__class__.__name__ if result.ui_component else None
            ),
        )
        await self.log_event(event)

    async def log_ui_feature_access(
        self,
        user: "User",
        feature_name: str,
        access_granted: bool,
        required_groups: List[str],
        conversation_id: str,
        request_id: str,
    ) -> None:
        """Convenience method for logging UI feature access checks.

        Args:
            user: User attempting to access the feature
            feature_name: Name of the UI feature
            access_granted: Whether access was granted
            required_groups: Groups required to access the feature
            conversation_id: Conversation identifier
            request_id: Request identifier
        """
        event = UiFeatureAccessCheckEvent(
            user_id=user.id,
            username=user.username,
            user_email=user.email,
            user_groups=user.group_memberships,
            conversation_id=conversation_id,
            request_id=request_id,
            feature_name=feature_name,
            access_granted=access_granted,
            required_groups=required_groups,
        )
```

```
"""
Audit event models.

This module contains data models for audit logging events.
"""

import uuid
from datetime import datetime
from typing import Any, Dict, List, Optional

from pydantic import BaseModel, Field

from .._compat import StrEnum


class AuditEventType(StrEnum):
    """Types of audit events."""

    # Access control events
    TOOL_ACCESS_CHECK = "tool_access_check"
    UI_FEATURE_ACCESS_CHECK = "ui_feature_access_check"

    # Tool execution events
    TOOL_INVOCATION = "tool_invocation"
    TOOL_RESULT = "tool_result"

    # Conversation events
    MESSAGE_RECEIVED = "message_received"
    AI_RESPONSE_GENERATED = "ai_response_generated"
    CONVERSATION_CREATED = "conversation_created"

    # Security events
```

```
    ACCESS_DENIED = "access_denied"
    AUTHENTICATION_ATTEMPT = "authentication_attempt"


class AuditEvent(BaseModel):
    """Base audit event with common fields."""

    event_id: str = Field(default_factory=lambda: str(uuid.uuid4()))
    event_type: AuditEventType
    timestamp: datetime = Field(default_factory=datetime.utcnow)

    # User context
    user_id: str
    username: Optional[str] = None
    user_email: Optional[str] = None
    user_groups: List[str] = Field(default_factory=list)

    # Request context
    conversation_id: str
    request_id: str
    remote_addr: Optional[str] = None

    # Event-specific data
    details: Dict[str, Any] = Field(default_factory=dict)

    # Privacy/redaction markers
    contains_pii: bool = False
    redacted_fields: List[str] = Field(default_factory=list)


class ToolAccessCheckEvent(AuditEvent):
    """Audit event for tool access permission checks."""

    event_type: AuditEventType = AuditEventType.TOOL_ACCESS_CHECK
    tool_name: str
    access_granted: bool
    required_groups: List[str] = Field(default_factory=list)
    reason: Optional[str] = None


class ToolInvocationEvent(AuditEvent):
    """Audit event for actual tool executions."""

    event_type: AuditEventType = AuditEventType.TOOL_INVOCATION
    tool_call_id: str
    tool_name: str

    # Parameters with sanitization support
    parameters: Dict[str, Any] = Field(default_factory=dict)
    parameters_sanitized: bool = False

    # UI context at invocation time
    ui_features_available: List[str] = Field(default_factory=list)


class ToolResultEvent(AuditEvent):
    """Audit event for tool execution results."""

    event_type: AuditEventType = AuditEventType.TOOL_RESULT
    tool_call_id: str
    tool_name: str
    success: bool
    error: Optional[str] = None
    execution_time_ms: float = 0.0

    # Result metadata (without full content for size)
    result_size_bytes: Optional[int] = None
    ui_component_type: Optional[str] = None
```

```
class UiFeatureAccessCheckEvent(AuditEvent):
    """Audit event for UI feature access checks."""

    event_type: AuditEventType = AuditEventType.UI_FEATURE_ACCESS_CHECK
    feature_name: str
    access_granted: bool
    required_groups: List[str] = Field(default_factory=list)


class AiResponseEvent(AuditEvent):
    """Audit event for AI-generated responses."""

    event_type: AuditEventType = AuditEventType.AI_RESPONSE_GENERATED

    # Response metadata
    response_length_chars: int
    response_length_tokens: Optional[int] = None

    # Full text (optional, configurable)
    response_text: Optional[str] = None
    response_hash: str  # SHA256 for integrity verification

    # Model info
    model_name: Optional[str] = None
    temperature: Optional[float] = None

    # Tool calls in response
    tool_calls_count: int = 0
    tool_names: List[str] = Field(default_factory=list)
```

Local logging-based AuditLogger (packaged integration)

```
"""
Local audit logger implementation using Python logging.

This module provides a simple audit logger that writes events using
the standard Python logging module, useful for development and testing.
"""

import json
import logging
from typing import Optional

from vanna.core.audit import AuditEvent, AuditLogger

logger = logging.getLogger(__name__)


class LoggingAuditLogger(AuditLogger):
    """Audit logger that writes events to Python logger as structured JSON.

    This implementation uses logger.info() to emit audit events as JSON,
    making them easy to parse and route to log aggregation systems.

    Example:
        audit_logger = LoggingAuditLogger()
        agent = Agent(
            llm_service=...,
            audit_logger=audit_logger
        )
    """

    def __init__(self, log_level: int = logging.INFO):
        """Initialize the logging audit logger.

        Args:
            log_level: Log level to use for audit events (default: INFO)
```

```
        """
        self.log_level = log_level

    async def log_event(self, event: AuditEvent) -> None:
        """Log an audit event as structured JSON.

        Args:
            event: The audit event to log
        """
        try:
            # Convert event to dict for JSON serialization
            event_dict = event.model_dump(mode="json", exclude_none=True)

            # Format as single-line JSON for easy parsing
            event_json = json.dumps(event_dict, separators=(",", ":"))

            # Log with structured prefix for easy filtering
            logger.log(
                self.log_level,
                f"[AUDIT] {event.event_type.value} | {event_json}",
            )
        except Exception as e:
            # Don't fail the operation if audit logging fails
            logger.error(f"Failed to log audit event: {e}", exc_info=True)
```

Recommended audit events

User authentication events (login, token refresh, logout).

Tool executions and parameters (redact secrets).

SQL execution metadata (query hash, duration, row count) and failures.

File system or Python tool usage (path accessed, command executed).

16. Python API Reference

This section is a practical guide to the main Python classes you ll use most often.

16.1 AgentConfig

AgentConfig controls limits, streaming, UI features, and audit logging. Excerpt:

```
"""
Agent configuration.

This module contains configuration models that control agent behavior.
"""

from typing import TYPE_CHECKING, Dict, List, Optional

from pydantic import BaseModel, Field

from .._compat import StrEnum

if TYPE_CHECKING:
    from ..user import User


class UiFeature(StrEnum):
    UI_FEATURE_SHOW_TOOL_NAMES = "tool_names"
    UI_FEATURE_SHOW_TOOL_ARGUMENTS = "tool_arguments"
    UI_FEATURE_SHOW_TOOL_ERROR = "tool_error"
    UI_FEATURE_SHOW_TOOL_INVOCATION_MESSAGE_IN_CHAT = "tool_invocation_message_in_chat"
    UI_FEATURE_SHOW_MEMORY_DETAILED_RESULTS = "memory_detailed_results"


# Optional: you can also define defaults if you want a shared baseline
DEFAULT_UI_FEATURES: Dict[str, List[str]] = {
    UiFeature.UI_FEATURE_SHOW_TOOL_NAMES: ["admin", "user"],
    UiFeature.UI_FEATURE_SHOW_TOOL_ARGUMENTS: ["admin"],
    UiFeature.UI_FEATURE_SHOW_TOOL_ERROR: ["admin"],
    UiFeature.UI_FEATURE_SHOW_TOOL_INVOCATION_MESSAGE_IN_CHAT: ["admin"],
    UiFeature.UI_FEATURE_SHOW_MEMORY_DETAILED_RESULTS: ["admin"],
}


class UiFeatures(BaseModel):
    """UI features with group-based access control using the same pattern as tools.

    Each field specifies which groups can access that UI feature.
    Empty list means the feature is accessible to all users.
    Uses the same intersection logic as tool access control.
    """

    # Custom features for extensibility
    feature_group_access: Dict[str, List[str]] = Field(
        default_factory=lambda: DEFAULT_UI_FEATURES.copy(),
        description="Which groups can access UI features",
    )

    def can_user_access_feature(self, feature_name: str, user: "User") -> bool:
        """Check if user can access a UI feature using same logic as tools.

        Args:
            feature_name: Name of the UI feature to check
            user: User object with group_memberships

        Returns:
            True if user has access, False otherwise
```

```
        """
        # Then try custom features
        if feature_name in self.feature_group_access:
            allowed_groups = self.feature_group_access[feature_name]
        else:
            # Feature doesn't exist, deny access
            return False

        # Empty list means all users can access (same as tools)
        if not allowed_groups:
            return True

        # Same intersection logic as tool access control
        user_groups = set(user.group_memberships)
        feature_groups = set(allowed_groups)
        return bool(user_groups & feature_groups)

    def register_feature(self, name: str, access_groups: List[str]) -> None:
        """Register a custom UI feature with group access control.

        Args:
            name: Name of the custom feature
            access_groups: List of groups that can access this feature
        """
        self.feature_group_access[name] = access_groups


class AuditConfig(BaseModel):
    """Configuration for audit logging."""

    enabled: bool = Field(default=True, description="Enable audit logging")
    log_tool_access_checks: bool = Field(
        default=True, description="Log tool access permission checks"
    )
    log_tool_invocations: bool = Field(
        default=True, description="Log tool invocations with parameters"
    )
    log_tool_results: bool = Field(
        default=True, description="Log tool execution results"
    )
    log_ui_feature_checks: bool = Field(
        default=False, description="Log UI feature access checks (can be noisy)"
    )
    log_ai_responses: bool = Field(
        default=True, description="Log AI-generated responses"
    )
    include_full_ai_responses: bool = Field(
        default=False,
        description="Include full AI response text in logs (privacy concern)",
    )
    sanitize_tool_parameters: bool = Field(
        default=True, description="Sanitize sensitive parameters (passwords, tokens)"
    )


class AgentConfig(BaseModel):
    """Configuration for agent behavior."""

    max_tool_iterations: int = Field(default=10, gt=0)
    stream_responses: bool = Field(default=True)
    auto_save_conversations: bool = Field(default=True)
    include_thinking_indicators: bool = Field(default=True)
    temperature: float = Field(default=0.7, ge=0.0, le=2.0)
    max_tokens: Optional[int] = Field(default=None, gt=0)
    ui_features: UiFeatures = Field(default_factory=UiFeatures)
    audit_config: AuditConfig = Field(default_factory=AuditConfig)
```

16.2 Agent

Agent.send_message() is async and yields UiComponent objects as the workflow progresses. You typically call it from a server route and stream the output to the client.

```
"""
Agent implementation for the Vanna Agents framework.

This module provides the main Agent class that orchestrates the interaction
between LLM services, tools, and conversation storage.
"""

import traceback
import uuid
from typing import TYPE_CHECKING, AsyncGenerator, List, Optional

from vanna.components import (
    UiComponent,
    SimpleTextComponent,
    RichTextComponent,
    StatusBarUpdateComponent,
    TaskTrackerUpdateComponent,
    ChatInputUpdateComponent,
    StatusCardComponent,
    Task,
)
from .config import AgentConfig
from vanna.core.storage import ConversationStore
from vanna.core.llm import LlmService
from vanna.core.system_prompt import SystemPromptBuilder
from vanna.core.storage import Conversation, Message
from vanna.core.llm import LlmMessage, LlmRequest, LlmResponse
from vanna.core.tool import ToolCall, ToolContext, ToolResult, ToolSchema
from vanna.core.user import User
from vanna.core.registry import ToolRegistry
from vanna.core.system_prompt import DefaultSystemPromptBuilder
from vanna.core.lifecycle import LifecycleHook
from vanna.core.middleware import LlmMiddleware
from vanna.core.workflow import WorkflowHandler, DefaultWorkflowHandler
from vanna.core.recovery import ErrorRecoveryStrategy, RecoveryActionType
from vanna.core.enricher import ToolContextEnricher
from vanna.core.enhancer import LlmContextEnhancer, DefaultLlmContextEnhancer
from vanna.core.filter import ConversationFilter
from vanna.core.observability import ObservabilityProvider
from vanna.core.user.resolver import UserResolver
from vanna.core.user.request_context import RequestContext
from vanna.core.agent.config import UiFeature
from vanna.core.audit import AuditLogger
from vanna.capabilities.agent_memory import AgentMemory

import logging

logger = logging.getLogger(__name__)

logger.info("Loaded vanna.core.agent.agent module")

if TYPE_CHECKING:
    pass


class Agent:
    """Main agent implementation.

    The Agent class orchestrates LLM interactions, tool execution, and conversation
    management. It provides 7 extensibility points for customization:

    - lifecycle_hooks: Hook into message and tool execution lifecycle
```

```
    - llm_middlewares: Intercept and transform LLM requests/responses
    - error_recovery_strategy: Handle errors with retry logic
    - context_enrichers: Add data to tool execution context
    - llm_context_enhancer: Enhance LLM system prompts and messages with context
    - conversation_filters: Filter conversation history before LLM calls
    - observability_provider: Collect telemetry and monitoring data

    Example:
        agent = Agent(
            llm_service=AnthropicLlmService(api_key="..."),
            tool_registry=registry,
            conversation_store=store,
            lifecycle_hooks=[QuotaCheckHook()],
            llm_middlewares=[CachingMiddleware()],
            llm_context_enhancer=DefaultLlmContextEnhancer(agent_memory),
            observability_provider=LoggingProvider()
        )
    """

    def __init__(
        self,
        llm_service: LlmService,
        tool_registry: ToolRegistry,
        user_resolver: UserResolver,
        agent_memory: AgentMemory,
        conversation_store: Optional[ConversationStore] = None,
        config: AgentConfig = AgentConfig(),
        system_prompt_builder: SystemPromptBuilder = DefaultSystemPromptBuilder(),
        lifecycle_hooks: List[LifecycleHook] = [],
        llm_middlewares: List[LlmMiddleware] = [],
        workflow_handler: Optional[WorkflowHandler] = None,
        error_recovery_strategy: Optional[ErrorRecoveryStrategy] = None,
        context_enrichers: List[ToolContextEnricher] = [],
        llm_context_enhancer: Optional[LlmContextEnhancer] = None,
        conversation_filters: List[ConversationFilter] = [],
        observability_provider: Optional[ObservabilityProvider] = None,
        audit_logger: Optional[AuditLogger] = None,
    ):
        self.llm_service = llm_service
        self.tool_registry = tool_registry
        self.user_resolver = user_resolver
        self.agent_memory = agent_memory

        # Import here to avoid circular dependency
        if conversation_store is None:
            from vanna.integrations.local import MemoryConversationStore

            conversation_store = MemoryConversationStore()

        self.conversation_store = conversation_store
        self.config = config
        self.system_prompt_builder = system_prompt_builder
        self.lifecycle_hooks = lifecycle_hooks
        self.llm_middlewares = llm_middlewares

        # Use DefaultWorkflowHandler if none provided
        if workflow_handler is None:
            workflow_handler = DefaultWorkflowHandler()
        self.workflow_handler = workflow_handler

        self.error_recovery_strategy = error_recovery_strategy
        self.context_enrichers = context_enrichers

        # Use DefaultLlmContextEnhancer if none provided
        if llm_context_enhancer is None:
            llm_context_enhancer = DefaultLlmContextEnhancer(agent_memory)
        self.llm_context_enhancer = llm_context_enhancer
```

```
        self.conversation_filters = conversation_filters
        self.observability_provider = observability_provider
        self.audit_logger = audit_logger

        # Wire audit logger into tool registry
        if self.audit_logger and self.config.audit_config.enabled:
            self.tool_registry.audit_logger = self.audit_logger
            self.tool_registry.audit_config = self.config.audit_config

        logger.info("Initialized Agent")

    async def send_message(
        self,
        request_context: RequestContext,
        message: str,
        *,
        conversation_id: Optional[str] = None,
    ) -> AsyncGenerator[UiComponent, None]:
        """
        Process a user message and yield UI components with error handling.

        Args:
            request_context: Request context for user resolution (includes metadata)
            message: User's message content
            conversation_id: Optional conversation ID; if None, creates new conversation

        Yields:
            UiComponent instances for UI updates
        """
        try:
            # Delegate to internal method
            async for component in self._send_message(
                request_context, message, conversation_id=conversation_id
            ):
                yield component
        except Exception as e:
            # Log full stack trace
            stack_trace = traceback.format_exc()
            logger.error(
                f"Error in send_message (conversation_id={conversation_id}): {e}\n{stack_trace}",
                exc_info=True,
            )

            # Log to observability provider if available
            if self.observability_provider:
                try:
                    error_span = await self.observability_provider.create_span(
                        "agent.send_message.error",
                        attributes={
                            "error_type": type(e).__name__,
                            "error_message": str(e),
                            "conversation_id": conversation_id or "none",
                        },
                    )
                    await self.observability_provider.end_span(error_span)
                    await self.observability_provider.record_metric(
                        "agent.error.count",
                        1.0,
                        "count",
                        tags={"error_type": type(e).__name__},
                    )
                except Exception as obs_error:
                    logger.error(
                        f"Failed to log error to observability provider: {obs_error}",
                        exc_info=True,
                    )

            # Yield error component to UI (simple, user-friendly message)
```

```
            error_description = "An unexpected error occurred while processing your message. Please try
            if conversation_id:
```

Send a message and stream UI components (pattern)

```
async for ui_component in agent.send_message(
    request_context=request_context,
    message="Show sales by region for the last 30 days",
    conversation_id="optional-conv-id",
):
    # Convert ui_component to JSON and stream to the client
    ...
```

16.3 Conversation stores

ConversationStore is an abstraction; the local integration includes memory and file-based stores.

MemoryConversationStore excerpt

```
"""
In-memory conversation store implementation.

This module provides a simple in-memory implementation of the ConversationStore
interface, useful for testing and development.
"""

from typing import Dict, List, Optional

from vanna.core.storage import ConversationStore, Conversation, Message
from vanna.core.user import User


class MemoryConversationStore(ConversationStore):
    """In-memory conversation store."""

    def __init__(self) -> None:
        self._conversations: Dict[str, Conversation] = {}

    async def create_conversation(
        self, conversation_id: str, user: User, initial_message: str
    ) -> Conversation:
        """Create a new conversation with the specified ID."""
        conversation = Conversation(
            id=conversation_id,
            user=user,
            messages=[Message(role="user", content=initial_message)],
        )
        self._conversations[conversation_id] = conversation
        return conversation

    async def get_conversation(
        self, conversation_id: str, user: User
    ) -> Optional[Conversation]:
        """Get conversation by ID, scoped to user."""
        conversation = self._conversations.get(conversation_id)
        if conversation and conversation.user.id == user.id:
            return conversation
        return None

    async def update_conversation(self, conversation: Conversation) -> None:
        """Update conversation with new messages."""
        self._conversations[conversation.id] = conversation

    async def delete_conversation(self, conversation_id: str, user: User) -> bool:
        """Delete conversation."""
```

```
        conversation = await self.get_conversation(conversation_id, user)
        if conversation:
            del self._conversations[conversation_id]
            return True
        return False

    async def list_conversations(
        self, user: User, limit: int = 50, offset: int = 0
    ) -> List[Conversation]:
        """List conversations for user."""
        user_conversations = [
            conv for conv in self._conversations.values() if conv.user.id == user.id
        ]
        # Sort by updated_at desc
        user_conversations.sort(key=lambda x: x.updated_at, reverse=True)
        return user_conversations[offset : offset + limit]
```

FileSystemConversationStore excerpt

```
"""
File system conversation store implementation.

This module provides a file-based implementation of the ConversationStore
interface that persists conversations to disk as a directory structure.
"""

import json
import os
from pathlib import Path
from typing import Dict, List, Optional
from datetime import datetime
import time

from vanna.core.storage import ConversationStore, Conversation, Message
from vanna.core.user import User


class FileSystemConversationStore(ConversationStore):
    """File system-based conversation store.

    Stores conversations as directories with individual message files:
    conversations/{conversation_id}/
        metadata.json - conversation metadata (id, user info, timestamps)
        messages/
            {timestamp}_{index}.json - individual message files
    """

    def __init__(self, base_dir: str = "conversations") -> None:
        """Initialize the file system conversation store.

        Args:
            base_dir: Base directory for storing conversations
        """
        self.base_dir = Path(base_dir)
        self.base_dir.mkdir(parents=True, exist_ok=True)

    def _get_conversation_dir(self, conversation_id: str) -> Path:
        """Get the directory path for a conversation."""
        return self.base_dir / conversation_id

    def _get_metadata_path(self, conversation_id: str) -> Path:
        """Get the metadata file path for a conversation."""
        return self._get_conversation_dir(conversation_id) / "metadata.json"

    def _get_messages_dir(self, conversation_id: str) -> Path:
        """Get the messages directory for a conversation."""
        return self._get_conversation_dir(conversation_id) / "messages"
```

```
    def _save_metadata(self, conversation: Conversation) -> None:
        """Save conversation metadata to disk."""
        conv_dir = self._get_conversation_dir(conversation.id)
        conv_dir.mkdir(parents=True, exist_ok=True)

        metadata = {
            "id": conversation.id,
            "user": conversation.user.model_dump(mode="json"),
            "created_at": conversation.created_at.isoformat(),
            "updated_at": conversation.updated_at.isoformat(),
        }

        metadata_path = self._get_metadata_path(conversation.id)
        with open(metadata_path, "w") as f:
            json.dump(metadata, f, indent=2)

    def _load_messages(self, conversation_id: str) -> List[Message]:
        """Load all messages for a conversation."""
        messages_dir = self._get_messages_dir(conversation_id)

        if not messages_dir.exists():
            return []

        messages = []
        # Sort message files by name (timestamp_index ensures correct order)
        message_files = sorted(messages_dir.glob("*.json"))

        for file_path in message_files:
            try:
                with open(file_path, "r") as f:
                    data = json.load(f)
                message = Message.model_validate(data)
                messages.append(message)
            except (json.JSONDecodeError, ValueError) as e:
                print(f"Failed to load message from {file_path}: {e}")
                continue

        return messages

    def _append_message(
        self, conversation_id: str, message: Message, index: int
    ) -> None:
        """Append a message to the conversation."""
        messages_dir = self._get_messages_dir(conversation_id)
        messages_dir.mkdir(parents=True, exist_ok=True)

        # Use timestamp + index to ensure unique, ordered filenames
        timestamp = int(time.time() * 1000000)  # microseconds
        filename = f"{timestamp}_{index:06d}.json"
        file_path = messages_dir / filename

        with open(file_path, "w") as f:
            json.dump(message.model_dump(mode="json"), f, indent=2)

    async def create_conversation(
        self, conversation_id: str, user: User, initial_message: str
    ) -> Conversation:
        """Create a new conversation with the specified ID."""
        conversation = Conversation(
            id=conversation_id,
            user=user,
            messages=[Message(role="user", content=initial_message)],
        )

        # Save metadata
        self._save_metadata(conversation)

        # Save initial message
```

```
        self._append_message(conversation_id, conversation.messages[0], 0)

        return conversation

    async def get_conversation(
        self, conversation_id: str, user: User
    ) -> Optional[Conversation]:
        """Get conversation by ID, scoped to user."""
        metadata_path = self._get_metadata_path(conversation_id)

        if not metadata_path.exists():
            return None

        try:
            # Load metadata
            with open(metadata_path, "r") as f:
                metadata = json.load(f)

            # Verify ownership
            if metadata["user"]["id"] != user.id:
                return None

            # Load all messages
            messages = self._load_messages(conversation_id)

            # Reconstruct conversation
            conversation = Conversation(
                id=metadata["id"],
                user=User.model_validate(metadata["user"]),
                messages=messages,
                created_at=datetime.fromisoformat(metadata["created_at"]),
                updated_at=datetime.fromisoformat(metadata["updated_at"]),
            )

            return conversation
        except (json.JSONDecodeError, ValueError, KeyError) as e:
            print(f"Failed to load conversation {conversation_id}: {e}")
            return None

    async def update_conversation(self, conversation: Conversation) -> None:
        """Update conversation with new messages."""
        # Update the updated_at timestamp
        conversation.updated_at = datetime.now()

        # Save updated metadata
        self._save_metadata(conversation)

        # Get existing messages count to determine new message indices
        existing_messages = self._load_messages(conversation.id)
        existing_count = len(existing_messages)

        # Only append new messages (ones not already saved)
        for i, message in enumerate(
```

16.4 File system sandbox

The LocalFileSystem integration provides a root directory that file tools are constrained to.

```
"""
Local file system implementation.

This module provides a local file system implementation with per-user isolation.
"""

import asyncio
import hashlib
from pathlib import Path
```

```
from typing import List, Optional

from vanna.capabilities.file_system import CommandResult, FileSearchMatch, FileSystem
from vanna.core.tool import ToolContext

MAX_SEARCH_FILE_BYTES = 1_000_000


class LocalFileSystem(FileSystem):
    """Local file system implementation with per-user isolation."""

    def __init__(self, working_directory: str = "."):
        """Initialize with a working directory.

        Args:
            working_directory: Base directory where user-specific folders will be created
        """
        self.working_directory = Path(working_directory)

    def _get_user_directory(self, context: ToolContext) -> Path:
        """Get the user-specific directory by hashing the user ID.

        Args:
            context: Tool context containing user information

        Returns:
            Path to the user-specific directory
        """
        # Hash the user ID to create a directory name
        user_hash = hashlib.sha256(context.user.id.encode()).hexdigest()[:16]
        user_dir = self.working_directory / user_hash

        # Create the directory if it doesn't exist
        user_dir.mkdir(parents=True, exist_ok=True)

        return user_dir

    def _resolve_path(self, path: str, context: ToolContext) -> Path:
        """Resolve a path relative to the user's directory.

        Args:
            path: Path relative to user directory
            context: Tool context containing user information

        Returns:
            Absolute path within user's directory
        """
        user_dir = self._get_user_directory(context)
        resolved = user_dir / path

        # Ensure the path is within the user's directory (prevent directory traversal)
        try:
            resolved.resolve().relative_to(user_dir.resolve())
        except ValueError:
            raise PermissionError(
                f"Access denied: path '{path}' is outside user directory"
            )

        return resolved

    async def list_files(self, directory: str, context: ToolContext) -> List[str]:
        """List files in a directory within the user's isolated space."""
        directory_path = self._resolve_path(directory, context)

        if not directory_path.exists():
            raise FileNotFoundError(f"Directory '{directory}' does not exist")

        if not directory_path.is_dir():
```

```
            raise NotADirectoryError(f"'{directory}' is not a directory")

        files = []
        for item in directory_path.iterdir():
            if item.is_file():
                files.append(item.name)

        return sorted(files)

    async def read_file(self, filename: str, context: ToolContext) -> str:
        """Read the contents of a file within the user's isolated space."""
        file_path = self._resolve_path(filename, context)

        if not file_path.exists():
            raise FileNotFoundError(f"File '{filename}' does not exist")

        if not file_path.is_file():
            raise IsADirectoryError(f"'{filename}' is a directory, not a file")

        return file_path.read_text(encoding="utf-8")

    async def write_file(
        self, filename: str, content: str, context: ToolContext, overwrite: bool = False
    ) -> None:
        """Write content to a file within the user's isolated space."""
        file_path = self._resolve_path(filename, context)

        # Create parent directories if they don't exist
        file_path.parent.mkdir(parents=True, exist_ok=True)

        if file_path.exists() and not overwrite:
            raise FileExistsError(
                f"File '{filename}' already exists. Use overwrite=True to replace it."
            )

        file_path.write_text(content, encoding="utf-8")

    async def exists(self, path: str, context: ToolContext) -> bool:
        """Check if a file or directory exists within the user's isolated space."""
        try:
            resolved_path = self._resolve_path(path, context)
            return resolved_path.exists()
        except PermissionError:
            return False

    async def is_directory(self, path: str, context: ToolContext) -> bool:
        """Check if a path is a directory within the user's isolated space."""
        try:
            resolved_path = self._resolve_path(path, context)
            return resolved_path.exists() and resolved_path.is_dir()
        except PermissionError:
            return False

    async def search_files(
        self,
        query: str,
        context: ToolContext,
        *,
        max_results: int = 20,
        include_content: bool = False,
    ) -> List[FileSearchMatch]:
        """Search for files within the user's isolated space."""

        trimmed_query = query.strip()
        if not trimmed_query:
            raise ValueError("Search query must not be empty")

        user_dir = self._get_user_directory(context)
```

```
        matches: List[FileSearchMatch] = []
        query_lower = trimmed_query.lower()

        for path in user_dir.rglob("*"):
            if len(matches) >= max_results:
                break

            if not path.is_file():
                continue

            relative_path = path.relative_to(user_dir).as_posix()
            include_entry = False
            snippet: Optional[str] = None

            if query_lower in path.name.lower():
                include_entry = True
                snippet = "[filename match]"

            content: Optional[str] = None
            if include_content:
                try:
                    size = path.stat().st_size
                except OSError:
                    if include_entry:
                        matches.append(
                            FileSearchMatch(path=relative_path, snippet=snippet)
                        )
                    continue

                if size <= MAX_SEARCH_FILE_BYTES:
                    try:
                        content = path.read_text(encoding="utf-8")
                    except (UnicodeDecodeError, OSError):
                        content = None
                elif not include_entry:
```

17. End-to-End Recipes

This section provides copy/paste-ready patterns that combine the concepts above into working deployments.

17.1 A production-ish SQL analytics agent (OpenAI + Postgres + memory + charts)

```
import os
from vanna import Agent, AgentConfig
from vanna.core.registry import ToolRegistry
from vanna.core.user import CookieEmailUserResolver
from vanna.integrations.openai import OpenAILlmService
from vanna.integrations.postgres import PostgresRunner
from vanna.integrations.local.agent_memory import DemoAgentMemory
from vanna.integrations.local import MemoryConversationStore, LocalFileSystem
from vanna.tools import RunSqlTool, VisualizeDataTool, SaveTextMemoryTool, SaveQuestionToolArgsTool, Sea

# LLM
llm = OpenAILlmService(api_key=os.environ["OPENAI_API_KEY"], model="gpt-4.1-mini")

# SQL + filesystem for artifacts/exports
sql_runner = PostgresRunner(connection_string=os.environ["POSTGRES_DSN"])
fs = LocalFileSystem("./vanna_data")

# Memory
agent_memory = DemoAgentMemory(max_items=50_000)

# Tools
registry = ToolRegistry()
registry.register(RunSqlTool(sql_runner=sql_runner, file_system=fs))
registry.register(VisualizeDataTool(file_system=fs))
registry.register(SaveTextMemoryTool(agent_memory=agent_memory))
registry.register(SaveQuestionToolArgsTool(agent_memory=agent_memory))
registry.register(SearchSavedCorrectToolUsesTool(agent_memory=agent_memory))

# User resolver (cookie-based)
user_resolver = CookieEmailUserResolver(cookie_name="vanna_user_email")

agent = Agent(
    llm_service=llm,
    tool_registry=registry,
    user_resolver=user_resolver,
    agent_memory=agent_memory,
    conversation_store=MemoryConversationStore(),
    config=AgentConfig(stream_responses=True),
)
```

17.2 Run it with FastAPI

```
from vanna.servers.fastapi import VannaFastAPIServer

server = VannaFastAPIServer(agent=agent, dev_mode=False)
server.run(host="0.0.0.0", port=8000)
```

17.3 Add a custom tool with permission gating

```
from pydantic import BaseModel, Field
from vanna.core.tool import Tool, ToolContext, ToolResult

class AdminEchoArgs(BaseModel):
    text: str = Field(description="Text to echo back")
```

```
class AdminEchoTool(Tool[AdminEchoArgs]):
    @property
    def name(self) -> str:
        return "admin_echo"
    @property
    def description(self) -> str:
        return "Echo text back (admin-only)."
    @property
    def required_groups(self) -> set[str]:
        return {"admin"}
    def get_args_schema(self):
        return AdminEchoArgs
    async def execute(self, context: ToolContext, args: AdminEchoArgs) -> ToolResult:
        return ToolResult(success=True, result_for_llm=args.text)
```

17.4 Evaluation and regression testing

Vanna includes an evaluation framework to compare agent variants and enforce expected outcomes. The packaged example shows datasets, evaluators, and runners.

```
"""
Evaluation System Example

This example demonstrates how to use the evaluation framework to test
and compare agents. Shows:
- Creating test cases programmatically
- Running evaluations with multiple evaluators
- Comparing agent variants (e.g., different LLMs)
- Generating reports

Usage:
    PYTHONPATH=. python vanna/examples/evaluation_example.py
"""

import asyncio
from vanna import Agent, MockLlmService, MemoryConversationStore, User
from vanna.core.evaluation import (
    EvaluationRunner,
    EvaluationDataset,
    TestCase,
    ExpectedOutcome,
    AgentVariant,
    TrajectoryEvaluator,
    OutputEvaluator,
    EfficiencyEvaluator,
)
from vanna.core.registry import ToolRegistry


def create_sample_dataset() -> EvaluationDataset:
    """Create a sample dataset for demonstration."""

    eval_user = User(
        id="eval_user", username="evaluator", email="eval@example.com", permissions=[]
    )

    test_cases = [
        TestCase(
            id="test_001",
            user=eval_user,
            message="Hello, how are you?",
            expected_outcome=ExpectedOutcome(
                final_answer_contains=["hello", "hi"],
                max_execution_time_ms=3000,
            ),
            metadata={"category": "greeting", "difficulty": "easy"},
        ),
```

```
        TestCase(
            id="test_002",
            user=eval_user,
            message="What can you help me with?",
            expected_outcome=ExpectedOutcome(
                final_answer_contains=["help", "assist"],
                max_execution_time_ms=3000,
            ),
            metadata={"category": "capabilities", "difficulty": "easy"},
        ),
        TestCase(
            id="test_003",
            user=eval_user,
            message="Explain quantum computing",
            expected_outcome=ExpectedOutcome(
                final_answer_contains=["quantum", "computing"],
                min_components=1,
                max_execution_time_ms=5000,
            ),
            metadata={"category": "explanation", "difficulty": "medium"},
        ),
    ]

    return EvaluationDataset(
        name="Demo Test Cases",
        test_cases=test_cases,
        description="Sample test cases for evaluation demo",
    )


def create_test_agent(name: str, response_content: str) -> Agent:
    """Create a test agent with mock LLM."""
    return Agent(
```

18. Appendix Topic Mapping to the Official Docs Links

You provided a list of Vanna web documentation links. This table maps each topic to the relevant sections of this PDF.

Official doc topic	URL	Where in this PDF
Auth	https://vanna.ai/docs/placeholder/auth	Section 10
Web UI	https://vanna.ai/docs/placeholder/web-ui	Section 8
Web UI customization	https://vanna.ai/docs/placeholder/web-ui/customization	Section 8 (Customization ideas)
Deployment (Flask)	https://vanna.ai/docs/placeholder/deployment/flask	Section 9.2
Deployment (FastAPI)	https://vanna.ai/docs/placeholder/deployment/fastapi	Section 9.1
Training	https://vanna.ai/docs/placeholder/training	Section 5
Built-in tools	https://vanna.ai/docs/placeholder/built-in-tools	Section 6
Custom tools	https://vanna.ai/docs/placeholder/custom-tools	Section 6 (Custom tool)
Workflow handlers	https://vanna.ai/docs/placeholder/workflow-handlers	Section 7
Charts	https://vanna.ai/docs/placeholder/charts	Section 11
Memory backends	https://vanna.ai/docs/placeholder/memory-backends	Section 5
LLM context enhancers	https://vanna.ai/docs/placeholder/llm-context-enhancers	Section 12
System prompts	https://vanna.ai/docs/placeholder/system-prompts	Section 12
UI features	https://vanna.ai/docs/placeholder/ui-features	Section 11
Configure Anthropic (other)	https://vanna.ai/docs/configure/anthropic/other	Section 3
Lifecycle hooks	https://vanna.ai/docs/placeholder/lifecycle-hooks	Section 13
LLM middlewares	https://vanna.ai/docs/placeholder/llm-middlewares	Section 13
Error recovery	https://vanna.ai/docs/placeholder/error-recovery	Section 14
Context enrichers	https://vanna.ai/docs/placeholder/context-enrichers	Section 12 (Context enrichers)
Conversation filters	https://vanna.ai/docs/placeholder/conversation-filters	Section 14
Observability	https://vanna.ai/docs/placeholder/observability	Section 15
Audit logging	https://vanna.ai/docs/placeholder/audit-logging	Section 15
Python	https://vanna.ai/docs/placeholder/python	Section 16

19. Appendix Packaged Examples Index

The vanna Python package ships an examples module. Use these as references when building your own agent.

Example script	What it demonstrates
__init__.py	Example package marker
__main__.py	Interactive example runner (select examples by name)
anthropic_quickstart.py	Anthropic example using AnthropicLlmService.
artifact_example.py	Creating and emitting ArtifactComponent UI elements
claude_sqlite_example.py	Claude example using the SQL query tool with the Chinook database.
coding_agent_example.py	Example coding agent using the vanna-agents framework.
custom_system_prompt_example.py	Example demonstrating custom system prompt builder with dependency injection.
default_workflow_handler_example.py	Example demonstrating the DefaultWorkflowHandler with setup health checking.
email_auth_example.py	Email authentication example for the Vanna Agents framework.
evaluation_example.py	Evaluation System Example
extensibility_example.py	Comprehensive example demonstrating all extensibility interfaces.
minimal_example.py	Minimal Claude + SQLite example ready for FastAPI.
mock_auth_example.py	Mock authentication example to verify user resolution is working.
mock_custom_tool.py	Mock example showing how to create and use custom tools.
mock_quickstart.py	Mock quickstart example for the Vanna Agents framework.
mock_quota_example.py	Mock quota-based agent example using Mock LLM service.
mock_rich_components_demo.py	Mock rich components demonstration example.
mock_sqlite_example.py	Mock example showing how to use the SQL query tool with the Chinook database.
openai_quickstart.py	OpenAI example using OpenAILlmService.
primitive_components_demo.py	Demonstration of the new primitive component system.
quota_lifecycle_example.py	Example demonstrating lifecycle hooks for user quota management.
visualization_example.py	Example demonstrating SQL query execution with automatic visualization.

Note on API drift

A few packaged examples appear to use older Agent constructor signatures. When that happens, update the examples by passing the required user_resolver and agent_memory arguments as shown in Section 17.

20. References

Primary sources

Vanna web documentation (placeholders): https://vanna.ai/docs/placeholder/

Anthropic configuration page: https://vanna.ai/docs/configure/anthropic/other

GitHub repository: https://github.com/vanna-ai/vanna

Repository examples (source): https://github.com/vanna-ai/vanna/tree/main/src/vanna/examples

Repository source tree: https://github.com/vanna-ai/vanna/tree/main/src

PyPI distribution: https://pypi.org/project/vanna/

Key code modules referenced in this PDF (package paths)

vanna/core/agent/agent.py main agent orchestrator

vanna/core/tool/base.py Tool / ToolResult / ToolContext

vanna/core/registry.py tool registration and access control

vanna/tools/run_sql.py SQL execution tool

vanna/tools/visualize_data.py chart tool

vanna/capabilities/agent_memory memory abstractions

vanna/servers/fastapi and vanna/servers/flask server deployments