Vanna AI: Comprehensive Developer Reference Guide

Executive Summary

Vanna 2.0 is a **user-aware AI agent framework** that connects Large Language Models (LLMs) to SQL databases, enabling natural language to SQL conversion with enterprise-grade security. This guide provides developers with complete specifications of all functions, requirements, deployment options, and best practices for building production-grade applications.

Part 1: Core Overview

1.1 What is Vanna?

Vanna is an open-source Python framework built on **Retrieval-Augmented Generation (RAG)** principles. It transforms the database querying experience by:

- Converting natural language questions into accurate SQL queries
- Learning from successful interactions through Tool Memory
- Enforcing user-based permissions and access controls
- Providing streaming responses with rich UI components
- Supporting enterprise security requirements

1.2 Core Architecture

The Vanna 2.0 architecture consists of **six key components**:

1. **Core Agent** — Orchestrates LLM interactions with tool execution loops
2. **Tool System** — Extensible tool registry with group-based access control
3. **Storage Layer** — Abstract interfaces for conversations, audit logs, observability
4. **User Management** — User resolution with group-based permissions (RBAC)
5. **LLM Services** — Pluggable integrations for multiple LLM providers
6. **Vector Store** — Storage and retrieval of training embeddings

1.3 Deployment Models

Model	Description	Data Location	Best For
Self-Hosted	Open-source Python package on your infrastructure	All data stays local	Maximum control, sensitive data, air-gapped environments

Model	Description	Data Location	Best For
Cloud Premium	Fully managed Vanna premium services	Vanna cloud infrastructure	Rapid deployment, managed observability
Hybrid	Python local with premium services for telemetry	Conversations local, telemetry in cloud	Balance between control and managed services

Part 2: Installation & Setup

2.1 System Requirements

- **Python Version**: 3.8 or higher (tested up to 3.12)
- **Operating System**: Windows, macOS, Linux (including Ubuntu 24.04.3 LTS)
- **RAM**: Minimum 4GB (8GB+ recommended for production)
- **Disk Space**: 2GB+ depending on vector store implementation

2.2 Installation Methods

Basic Installation

```
pip install vanna
```

With Database Support

```
# PostgreSQL
pip install 'vanna[postgres]'

# MySQL
pip install 'vanna[mysql]'

# Microsoft SQL Server
pip install 'vanna[mssql]'

# BigQuery
pip install 'vanna[bigquery]'

# Snowflake
pip install 'vanna[snowflake]'

# All databases
pip install 'vanna[all-databases]'
```

With Vector Store Support

```
# ChromaDB (default, lightweight)
pip install 'vanna[chroma]'

# Qdrant
pip install 'vanna[qdrant]'

# Milvus
pip install 'vanna[milvus]'

# Pinecone
pip install 'vanna[pinecone]'
```

2.3 Core Dependencies

Essential Python packages automatically installed:

- `pandas` — Data manipulation and DataFrame operations
- `requests` — HTTP client for API calls
- `pydantic` — Data validation using Python type annotations
- `fastapi` — Web framework for API deployment
- `sqlalchemy` — Database abstraction layer
- `plotly` — Data visualization library
- `openai` — OpenAI API client (if using OpenAI models)

Part 3: Core Functions & Methods

3.1 Function Nomenclature

Vanna uses a consistent naming convention to indicate function behavior:

Prefix	Definition	Examples
`vn.set_`	Sets a session variable	`set_model()`, `set_api_key()`
`vn.get_`	Performs read-only operations	`get_model()`, `get_training_data()`
`vn.add_`	Adds content to the model	`add_sql()`, `add_ddl()`, `add_documentation()`
`vn.generate_`	Generates AI-based output	`generate_sql()`, `generate_plotly_code()`
`vn.run_`	Executes code (SQL or Plotly)	`run_sql()`, `run_plotly_code()`
`vn.remove_`	Removes training data	`remove_training_data()`
`vn.connect_`	Connects to a database	`connect_to_postgres()`, `connect_to_bigquery()`
`vn.train()`	Trains the model with new data	`train()` (wrapper for add_* methods)

3.2 Initialization & Configuration

Basic Setup

```
from vanna import Agent, AgentConfig
from vanna.integrations.anthropic import AnthropicLlmService
from vanna.core.registry import ToolRegistry
from vanna.core.user import UserResolver, User, RequestContext

# Step 1: Configure LLM
llm = AnthropicLlmService(
    model='claude-3-sonnet-20240229',
    api_key='your-anthropic-api-key'
)

# Step 2: Initialize Tool Registry
tool_registry = ToolRegistry()

# Step 3: Create User Resolver
class SimpleUserResolver(UserResolver):
    async def resolve_user(self, request_context: RequestContext) -&gt; User:
        return User(
            id="default_user",
            username="developer",
            group_memberships=['user']
        )

# Step 4: Create Agent
agent = Agent(
    llm_service=llm,
    tool_registry=tool_registry,
    user_resolver=SimpleUserResolver()
)
```

Configuration Options

```
config = AgentConfig(
    max_tool_iterations=10,           # Max tool calls per message
    stream_responses=True,            # Enable streaming responses
    temperature=0.7,                  # LLM creativity (0-1)
    include_thinking_indicators=True, # Show "Thinking..." states
    auto_save_conversations=True,     # Auto-persist conversations
    max_tokens=None                   # Maximum response tokens
)
```

3.3 Database Connection Functions

PostgreSQL Connection

```
vn.connect_to_postgres(
    host='localhost',
    dbname='your_database',
    user='postgres',
    password='password',
    port=5432
)
```

MySQL Connection

```
vn.connect_to_mysql(
    host='localhost',
    dbname='your_database',
    user='root',
    password='password',
    port=3306
)
```

Microsoft SQL Server Connection

```
vn.connect_to_mssql(
    odbc_conn_str='Driver={ODBC Driver 17 for SQL Server};Server=server_name;Database=db_
)
```

SQLite Connection

```
vn.connect_to_sqlite(url='path/to/database.sqlite')
```

BigQuery Connection

```
vn.connect_to_bigquery(
    project_id='your-gcp-project',
    cred_file_path='path/to/credentials.json'
)
```

Snowflake Connection

```
vn.connect_to_snowflake(
    account='your_account',
    user='your_user',
    password='your_password',
    warehouse='COMPUTE_WH',
    database='YOUR_DB',
    schema='PUBLIC'
)
```

DuckDB Connection

```
vn.connect_to_duckdb(
    url=':memory:',  # or 'path/to/file.duckdb'
    init_sql=None
)
```

Oracle Connection

```
vn.connect_to_oracle(
    user='your_user',
    password='your_password',
    dsn='host:port/sid'
)
```

3.4 Training Functions

Training equips Vanna with knowledge about your database structure and business logic.

Training with DDL (Data Definition Language)

```
# Add a single DDL statement
vn.train(ddl="""
    CREATE TABLE IF NOT EXISTS customers (
        customer_id INT PRIMARY KEY,
        first_name VARCHAR(50),
        last_name VARCHAR(50),
        email VARCHAR(100),
        registration_date DATE
    )
""")

# Or add multiple DDL statements
vn.add_ddl(ddl="CREATE TABLE orders (...)")
```

Training with Documentation

```
# Add business logic documentation
vn.train(documentation="""
    The 'customers' table contains all registered users.
    'registration_date' uses UTC timezone.
    Null emails indicate anonymous accounts.
""")
```

```
# Or direct addition
vn.add_documentation(doc="Business context explanation")
```

Training with SQL Examples

```
# Add question-SQL pairs
vn.train(sql="""
    SELECT customer_id, email, registration_date
    FROM customers
    WHERE registration_date &gt; CURRENT_DATE - INTERVAL 30 DAY
    /* This query retrieves customers registered in the last 30 days */
""")

# Or add with question
vn.add_question_sql(
    question="What customers registered in the last 30 days?",
    sql="SELECT customer_id, email FROM customers WHERE registration_date &gt; CURRENT_DA
)
```

Automatic Training from Schema

```
# Extract schema information automatically
df_schema = vn.run_sql("SELECT * FROM INFORMATION_SCHEMA.COLUMNS")
plan = vn.get_training_plan_generic(df_schema)
vn.train(plan=plan)
```

3.5 Query Generation Functions

Generate SQL from Natural Language

```
# Primary method: generate_sql()
sql_query = vn.generate_sql(
    question="What are the top 10 customers by total orders?"
)
print(sql_query)
# Output: SELECT customer_id, COUNT(*) as order_count FROM orders GROUP BY customer_id OR
```

Execute SQL Queries

```
# Run the generated SQL
result_dataframe = vn.run_sql(sql_query)
print(result_dataframe)
```

Generate Visualizations

```
# Generate Plotly code for charts
plotly_code = vn.generate_plotly_code(
    question="Show sales by region",
    sql=sql_query,
    df=result_dataframe
)

# Execute the visualization code
figure = vn.get_plotly_figure(
    plotly_code=plotly_code,
    df=result_dataframe
)
figure.show()
```

Generate Explanations

```
# Generate natural language explanation
explanation = vn.generate_explanation(
    sql=sql_query
)
print(explanation)
```

Generate Follow-up Questions

```
# Generate contextually relevant follow-up questions
followup_questions = vn.generate_followup_questions(
    question="What are the top 10 customers?",
    sql=sql_query,
    df=result_dataframe
)
for q in followup_questions:
    print(f"- {q}")
```

3.6 Convenience Method: ask()

The `ask()` method combines all steps into a single function:

```
result = vn.ask(
    question="What is my total revenue?",
    visualize=True,  # Generate charts
    log_sql=True     # Log the generated SQL
)

# Returns dictionary with:
# {
#     'sql': 'SELECT SUM(amount) FROM orders',
#     'df': &lt;DataFrame&gt;,
#     'figure': &lt;Plotly Figure&gt;,
#     'explanation': 'This query...',
#     'followup_questions': ['Which region?', ...]
# }
```

3.7 Training Data Management

Retrieve Training Data

```
# Get all training data
all_training = vn.get_training_data()

# Returns DataFrame with columns: id, type (DDL/SQL/Documentation), content
```

Retrieve Related Training Data

```
# Get DDL relevant to a question
related_ddl = vn.get_related_ddl(question="List active customers")

# Get relevant documentation
related_docs = vn.get_related_documentation(question="What is a VIP customer?")

# Get similar SQL examples
similar_sql = vn.get_similar_question_sql(question="Show top sellers")
```

Remove Training Data

```
# Remove training data by ID
vn.remove_training_data(id='training_id_123')

# Remove all training for a model (use cautiously)
vn.remove_training_data(id=None)  # Only if specifically implemented
```

Part 4: Advanced Features

4.1 Custom Tools Implementation

Create custom tools by extending the Tool base class:

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

# 1. Define argument schema
class EmailToolArgs(BaseModel):
    recipient: str = Field(description="Email address")
    subject: str = Field(description="Email subject")
    body: str = Field(description="Email body")

# 2. Implement the tool
class EmailTool(Tool[EmailToolArgs]):
    @property
    def name(self) -&gt; str:
        return "send_email"

    @property
    def description(self) -&gt; str:
        return "Send an email notification"

    @property
    def access_groups(self) -&gt; list[str]:
        return ['admin']  # Only admins can use

    def get_args_schema(self) -&gt; Type[EmailToolArgs]:
        return EmailToolArgs

    async def execute(self, context: ToolContext, args: EmailToolArgs) -&gt; ToolResult:
        # Implement logic
        success = await self.send_email(args.recipient, args.subject, args.body)

        return ToolResult(
            success=success,
            result_for_llm=f"Email sent to {args.recipient}",
            ui_component=UiComponent(
                rich_component=None,
                simple_component=SimpleTextComponent(text="Email sent successfully")
            ),
            metadata={"email": args.recipient}
        )

# 3. Register the tool
tool_registry.register(EmailTool())
```

4.2 Authentication & Permissions

JWT-Based Authentication

```
import jwt
from vanna.core.user import UserResolver, User, RequestContext

class JwtUserResolver(UserResolver):
    def __init__(self, secret_key: str):
        self.secret_key = secret_key

    async def resolve_user(self, request_context: RequestContext) -&gt; User:
        auth_header = request_context.get_header('Authorization')
        if not auth_header or not auth_header.startswith('Bearer '):
            return User(id="anonymous", username="guest")

        token = auth_header.split(' ')[^1]
        try:
            claims = jwt.decode(token, self.secret_key, algorithms=['HS256'])
            return User(
                id=claims['user_id'],
                username=claims['username'],
                email=claims['email'],
                group_memberships=claims.get('groups', [])
            )
        except jwt.InvalidTokenError:
            return User(id="anonymous", username="guest")
```

Role-Based Access Control

```
# Register tool with group restrictions
class AdminOnlyTool(Tool):
    @property
    def access_groups(self) -&gt; list[str]:
        return ['admin']  # Only users in 'admin' group

# Tool execution is automatically restricted
```

4.3 Lifecycle Hooks

Lifecycle hooks allow you to intercept and modify behavior at key points:

```
from vanna.core.lifecycle import LifecycleHook

class QuotaCheckHook(LifecycleHook):
    async def before_message(self, user: User, message: str) -&gt; str:
        # Check if user has quota remaining
        if not await self.check_quota(user.id):
            raise Exception("Quota exceeded")
        return message

    async def after_tool(self, result: ToolResult) -&gt; ToolResult:
        # Log tool execution
        await self.log_tool_execution(result)
        return result

# Register the hook
agent = Agent(
    llm_service=llm,
    tool_registry=tool_registry,
    user_resolver=user_resolver,
    lifecycle_hooks=[QuotaCheckHook()]
)
```

4.4 Observability & Monitoring

```
from vanna.core.observability import ObservabilityProvider

class LoggingProvider(ObservabilityProvider):
    async def create_span(self, name: str, attributes: dict):
        print(f"Starting: {name} with {attributes}")
        return Span(name, attributes)

    async def record_metric(self, name: str, value: float, unit: str, tags: dict):
        print(f"Metric: {name} = {value}{unit}")

agent = Agent(
    llm_service=llm,
    tool_registry=tool_registry,
    user_resolver=user_resolver,
    observability_provider=LoggingProvider()
)
```

Part 5: Vector Store Options

5.1 ChromaDB (Default)

Lightweight, in-memory vector store. Best for development and small deployments.

```
# Installation
pip install vanna  # ChromaDB included by default

# Configuration
class MyVanna(ChromaDB_VectorStore, OpenAI_Chat):
    def __init__(self, config=None):
        ChromaDB_VectorStore.__init__(self, config=config)
        OpenAI_Chat.__init__(self, config=config)

vn = MyVanna(config={'api_key': 'sk-...'})
```

5.2 Qdrant

Advanced vector database with production-ready features.

```
pip install 'vanna[qdrant]'

from vanna.qdrant import Qdrant_VectorStore
from qdrant_client import QdrantClient

class MyVanna(Qdrant_VectorStore, OpenAI_Chat):
    def __init__(self, config=None):
        Qdrant_VectorStore.__init__(self, config=config)
        OpenAI_Chat.__init__(self, config=config)
```

```
client = QdrantClient(":memory:")  # or "http://localhost:6333"
vn = MyVanna(config={'client': client, 'api_key': 'sk-...'})
```

5.3 Milvus

Highly scalable vector database with distributed architecture.

```
pip install 'vanna[milvus]'

from pymilvus import MilvusClient
from vanna.milvus import Milvus_VectorStore

class MyVanna(Milvus_VectorStore, OpenAI_Chat):
    def __init__(self, config=None):
        Milvus_VectorStore.__init__(self, config=config)
        OpenAI_Chat.__init__(self, config=config)

client = MilvusClient(uri="http://localhost:19530")
vn = MyVanna(config={'client': client, 'api_key': 'sk-...'})
```

Part 6: LLM Integration

6.1 Supported LLMs

OpenAI

```
from vanna.integrations.openai import OpenAI_Chat

# Configuration via API key or environment variable
vn.set_api_key('sk-your-key')
vn.set_model('gpt-4')
```

Anthropic Claude

```
from vanna.integrations.anthropic import AnthropicLlmService

llm = AnthropicLlmService(
    model='claude-3-sonnet-20240229',
    api_key='your-anthropic-key'
)
```

Ollama (Local)

```
from vanna.ollama import Ollama

vn = Ollama(config={
    'model': 'llama2:7b-chat',
    'ollama_host': 'http://localhost:11434'
})
```

Google Gemini

```
from vanna.integrations.google import GoogleLlmService

llm = GoogleLlmService(
    model='gemini-pro',
    api_key='your-gemini-key'
)
```

Mistral

```
from vanna.integrations.mistral import MistralLlmService

llm = MistralLlmService(
    model='mistral-large',
    api_key='your-mistral-key'
)
```

6.2 Custom LLM Implementation

```
from vanna.base import VannaBase

class MyCustomLLM(VannaBase):
    def __init__(self, config=None):
        super().__init__(config=config)

    def submit_prompt(self, prompt, **kwargs) -&gt; str:
        # Implement your LLM call
        response = self.call_my_model(prompt)
        return response
```

Part 7: Security & Best Practices

7.1 Security Features

Data Privacy

- Database contents not sent to LLM by default
- Set `allow_llm_to_see_data=False` for maximum privacy
- Use local deployments for sensitive data

Access Control

- Group-based permissions for tools and UI features

- User-scoped SQL execution with automatic filtering
- Row-level security support

Audit Logging

- Automatic parameter sanitization
- Tool access logging with timestamps
- Failed attempt tracking

7.2 Configuration Best Practices

```
# ✅ Secure Production Configuration
config = AgentConfig(
    stream_responses=True,
    auto_save_conversations=True,
    max_tool_iterations=5,  # Prevent infinite loops
    rate_limit_per_user=100,  # Daily limit
    enable_audit_logging=True
)

# ✅ Use environment variables for credentials
import os
api_key = os.getenv('VANNA_API_KEY')
db_password = os.getenv('DB_PASSWORD')

# ✅ Validate user inputs
def validate_question(question: str) -&gt; bool:
    max_length = 500
    return len(question) &lt;= max_length and question.strip()

# ✅ Implement error handling
try:
    result = vn.ask(question)
except Exception as e:
    logger.error(f"Query failed: {e}")
    return {"error": "Unable to process query"}
```

7.3 Common Vulnerabilities

Prompt Injection (CVE-2024-5565)

- Issue: Specially crafted prompts can execute arbitrary code
- Mitigation: Always validate user input, use parameter sanitization
- Update to latest version with security patches

Hallucination Prevention

```
# Provide comprehensive training data
vn.train(ddl="CREATE TABLE...", documentation="...", sql="...")

# Use function RAG for more deterministic outputs
# Verify generated SQL before execution
sql = vn.generate_sql(question)
validate_sql_syntax(sql)  # Add validation function
```

Part 8: Deployment Patterns

8.1 FastAPI Web Application

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

# Create agent (configured above)
agent = Agent(llm_service=llm, tool_registry=tools, user_resolver=resolver)

# Create FastAPI server
server = VannaFastAPIServer(agent)
app = server.create_app()

# Add custom routes
@app.get("/health")
async def health_check():
    return {"status": "healthy"}

# Run: uvicorn main:app --host 0.0.0.0 --port 8000
```

8.2 Streamlit Application

```
import streamlit as st
from vanna import Agent

st.set_page_config(page_title="Data Chat")

@st.cache_resource
def setup_agent():
    agent = Agent(
        llm_service=llm,
        tool_registry=tool_registry,
        user_resolver=user_resolver
    )
    agent.connect_to_postgres(...)
    return agent

agent = setup_agent()

st.title("□ Data Assistant")
question = st.text_input("Ask a question about your data:")

if question:
    with st.spinner("Thinking..."):
        result = agent.ask(question)
    st.dataframe(result['df'])
    if result['figure']:
        st.plotly_chart(result['figure'])
```

8.3 Docker Deployment

```
FROM python:3.10-slim

WORKDIR /app

# Install dependencies
COPY requirements.txt .
RUN pip install -r requirements.txt

# Copy application
COPY . .

# Expose port
EXPOSE 8000

# Run application
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

```
# requirements.txt
vanna&gt;=2.0.0
fastapi&gt;=0.104.0
uvicorn&gt;=0.24.0
python-dotenv&gt;=1.0.0
```

Part 9: Troubleshooting & Common Issues

9.1 Common Problems

Problem	Cause	Solution
Generated SQL is incorrect	Insufficient training data	Add more DDL, documentation, and example queries
Connection refused	Database not running	Verify database service is running and credentials correct
LLM timeout	Rate limiting	Implement retry logic with exponential backoff
Memory usage high	Large vector database	Consider pagination or use cloud vector store
Generated answers instead of SQL	Conversation context too long	Reset conversation or use context windows

9.2 Debugging

```
import logging

# Enable debug logging
logging.basicConfig(level=logging.DEBUG)
logger = logging.getLogger('vanna')

# Log all queries
result = vn.ask(question, log_sql=True)
print(f"Generated SQL: {result['sql']}")
print(f"Execution time: {result.get('duration', 'N/A')}")

# Validate connection
try:
    test_df = vn.run_sql("SELECT 1")
    print("Database connection OK")
except Exception as e:
    print(f"Database error: {e}")
```

Part 10: Performance Optimization

10.1 Query Optimization

```
# Limit training data size
# Use specific schema sections instead of full schema
vn.train(ddl="""
    CREATE TABLE large_table (
        id INT PRIMARY KEY,
        important_column VARCHAR(255),
        ...
    )
""")

# Optimize vector search
# Fewer, higher-quality examples beat many mediocre ones
```

10.2 Caching Strategies

```
from functools import lru_cache

@lru_cache(maxsize=128)
def get_schema_info():
    return vn.run_sql("SELECT * FROM INFORMATION_SCHEMA.COLUMNS")

# This caches expensive schema queries
```

Part 11: Version Migration

From Vanna 0.x to 2.0

Breaking Changes:

- Architecture completely redesigned
- Function names and signatures changed
- Vector store implementations vary
- LLM integration modernized

Migration Steps:

1. Backup all training data
2. Update installation: `pip install --upgrade vanna`
3. Refactor initialization code using new Agent/Tool patterns
4. Retrain models with new architecture
5. Test thoroughly in staging environment

Conclusion

Vanna 2.0 provides a production-ready framework for building AI-powered data applications. The modular architecture allows developers to:

- Choose their LLM provider and vector store
- Implement custom authentication and authorization
- Build enterprise-grade applications with audit logging
- Deploy flexibly (self-hosted, cloud, or hybrid)

For latest updates and community support, visit:

- **GitHub**: https://github.com/vanna-ai/vanna
- **Documentation**: https://vanna.ai/docs
- **Issues**: https://github.com/vanna-ai/vanna/issues

Document Version: 1.0
Last Updated: November 2025
Vanna Version: 2.0+
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37]

⁂

1. https://openaiagent.io/aitool/vanna-ai/
2. https://ask.vanna.ai/docs/base/

3. https://github.com/vanna-ai/vanna/issues/531
4. https://vanna.ai/docs/placeholder/auth
5. https://ask.vanna.ai/docs/postgres-openai-vanna-vannadb.html
6. https://vulnera.com/newswire/prompt-injection-vulnerability-in-vanna-ai-library-poses-risk-of-remote-code-execution-attacks/
7. https://ask.vanna.ai/docs/sqlite-openai-vanna-other-vectordb.html
8. https://github.com/vanna-ai/vanna/discussions/576
9. https://www.veracode.com/security/error-handling-flaws-information-and-how-fix-tutorial/
10. https://postgres.hashnode.dev/co-pilot-for-db-ops-using-vannaai-text2sql
11. https://skywork.ai/skypage/en/Vanna-Your-AI-Sidekick-for-Chatting-with-SQL-Databases/1975224980131082240
12. https://stackoverflow.com/questions/10333814/tell-pip-to-install-the-dependencies-of-packages-listed-in-a-requirement-file
13. https://mygit.osfipin.com/repository/640317662
14. https://milvus.io/docs/integrate_with_vanna.md
15. https://www.scribd.com/document/884968188/libraries-installed-in-vanna-virtual-environment
16. https://www.truefoundry.com/blog/ai-agent-registry
17. https://agixtech.com/chroma-vs-milvus-vs-qdrant-vector-db-comparison/
18. https://www.buildfastwithai.com/blogs/vanna-ai-turning-words-into-sql-magic-for-effortless-data-analysis
19. https://vanna.ai/docs/migration
20. https://qdrant.tech/documentation/frameworks/vanna-ai/
21. https://vanna.ai/docs/placeholder/python
22. https://devblogs.microsoft.com/azure-sql/vanna-ai-and-azure-sql-database/
23. https://ask.vanna.ai/blog/functionrag.html
24. https://vanna.ai/docs/placeholder/custom-tools
25. https://dev.to/selvapal/chat-with-your-sql-database-kej
26. https://zilliz.com/product/integrations/vanna-ai
27. https://www.youtube.com/watch?v=Ue_OUjJQnMw
28. https://github.com/vanna-ai/vanna/issues/754
29. https://www.marktechpost.com/2024/01/20/meet-vanna-an-open-source-python-rag-retrieval-augmented-generation-framework-for-sql-generation/
30. https://www.kdjingpai.com/en/vanna-bendebushuan/
31. https://github.com/vanna-ai/vanna
32. https://vanna.ai/data-security
33. https://try.vanna.ai/docs/vanna.html
34. https://milvus.io/docs/v2.4.x/integrate_with_vanna.md
35. https://www.linkedin.com/posts/balaram-panigrahy-2a7988210_texttosql-opensource-datasciencetools-activity-7324825775193346049-V6XD
36. https://www.linkedin.com/pulse/vanna-ai-revolutionizing-development-open-source-framework-barros-xeztf
37. https://try.vanna.ai/docs/other-database-other-llm-vannadb/