SliceBuddy

Technical Guide & File Reference

Version: v2 • Generated: 2026-02-20

This document explains how the SliceBuddy project is built: architecture, workflow, nodes, and what each file does. It’s written for programmers who want to understand the codebase quickly.

Core idea: deterministic planning first; LLM only for explanation (grounded with RAG).

Project overview

SliceBuddy takes a usage description + an STL file, extracts geometry signals, runs a rule-based planning pipeline (material, orientation, slicer settings, risks), and optionally uses an LLM to produce a clear human explanation.

Two outputs: (1) structured JSON plan (for UI/automation) and (2) a readable explanation (for humans).

Key concepts used

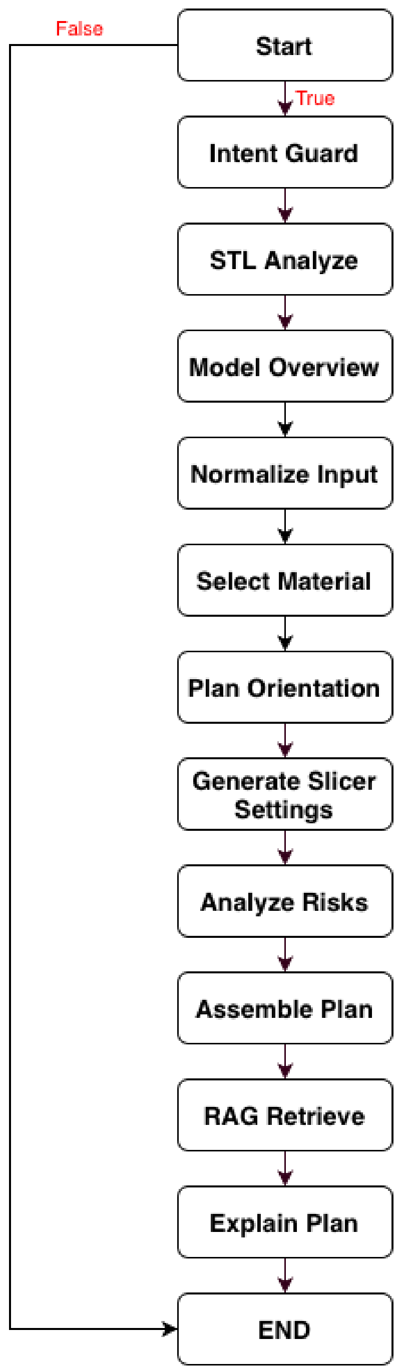
Concept	How SliceBuddy uses it
LangGraph	A graph-based workflow engine. Nodes read/write a shared state. You wire edges and can add conditional branching.
LangChain	Glue code and integrations (LLM wrapper, document types, text splitters, vector stores).
RAG (Retrieval-Augmented Generation)	Before calling the LLM, retrieve relevant knowledge snippets and include them as context so the LLM stays grounded.
Chroma	Local persistent vector database used to store embeddings of your Markdown knowledge base.
Embeddings	A numeric representation of text used for similarity search (here via OpenAI embeddings).
Trimesh + geometry heuristics	Used to extract STL signals like bounding box, contact area, overhang metrics, and mesh integrity flags.

LangGraph workflow

The workflow starts at START, immediately runs an input gate (INTENT_GUARD), then follows a mostly linear planning chain. There are two built-in conditionals:

1) **Stop branch**: if INTENT_GUARD decides the request is invalid, the graph ends early.
2) **Optional LLM explainer branch**: if USE_LLM_EXPLAINER=true, the graph appends RAG + LLM explanation before ending.

Visual overview



Node-by-node explanation

Each node is designed to do one job and communicate via the shared PlanState. This keeps decisions debuggable: you can print the state after each node and see exactly why the final plan looks the way it does.

INTENT_GUARD

Purpose: decide whether the input is a real print-planning request.
Reads: description, height_mm, width_mm, stl_path.
Writes: stop, and if stopping: plan + plan_explanation.
Current logic is minimal (length + dims or STL). If you want it to catch gibberish, add a heuristic here and append a warning or force stop=True.

STL_ANALYZE

Purpose: extract geometry signals from the STL (bbox, contact area, overhangs, mesh health).
Reads: stl_path.
Writes: stl_features and also back-fills height_mm/width_mm from the STL bbox so the rest of the pipeline works without manual dimensions.

MODEL_OVERVIEW

Purpose: create a beginner-friendly summary of what the model likely is and how "printable" it looks (without dumping raw numbers).
Reads: description, stl_features.
Writes: model_overview (used by CLI/UI for quick context).

NORMALIZE_INPUT

Purpose: normalize and validate raw inputs into a predictable shape.
Reads: description, height_mm, width_mm.
Writes: input_raw (snapshot), input_norm (cleaned), plus assumptions and warnings.

SELECT_MATERIAL

Purpose: choose a recommended filament using keywords + size hints.
Reads: input_norm (mainly description/size).
Writes: material (recommended + reason + alternatives) and may append warnings (e.g., ABS/ASA enclosure note).

PLAN_ORIENTATION

Purpose: propose an orientation that maximizes success (adhesion + stability) using STL-derived aspect/footprint when available.
Reads: input_norm, stl_features.
Writes: orientation including recommended orientation, reasons, trade-offs, and adhesion tips.

GENERATE_SLICER_SETTINGS

Purpose: generate practical slicer settings (walls, infill, supports, brim) from rule-based heuristics.
Reads: material, input_norm, stl_features.
Writes: slicer_settings and may append warnings (e.g., small contact area → add brim).

ANALYZE_RISKS

Purpose: turn geometry + settings into a structured risk report with mitigations (no auto-fixes).
Reads: slicer_settings, material, input_norm, stl_features.
Writes: risks (summary + items + mitigations) and may add warnings/assumptions.

ASSEMBLE_PLAN

Purpose: assemble the final structured plan object from the pieces produced by earlier nodes.
Reads: material/orientation/slicer_settings/risks/etc.
Writes: plan (a single dictionary used by UI/CLI). In your code this was originally DUMMY_PLAN but it's the right place to aggregate output.

RAG_RETRIEVE (optional)

Purpose: fetch relevant guidance from your Markdown knowledge base to ground the explanation.
Runs only if USE_LLM_EXPLAINER=true.
Reads: description, dims, and other hints to build a query.
Writes: rag_context (snippets) + rag_sources.

EXPLAIN_PLAN_LLM (optional)

Purpose: convert the structured plan + risks/warnings into a readable explanation for humans.
Runs only if USE_LLM_EXPLAINER=true.
Reads: plan, warnings, risks, and rag_context.
Writes: plan_explanation (plus optional model checks). Important: the LLM is used for wording and clarity, not for making the core decisions.

File reference

Below is a compact explanation of what each file does, organized by folder. This is meant to be a quick navigation map while you're coding.

app

File	What it does
`app/__init__.py`	Marks **app** as a Python package (empty).
`app/main.py`	FastAPI server exposing POST /plan: saves uploaded STL to a temp file, runs the LangGraph workflow, and returns the UI-friendly JSON payload (with CORS enabled for the Next.js dev server).

core

File	What it does
`core/__init__.py`	Marks **core** as a Python package (empty).
`core/config.py`	Loads environment variables from .env and provides a helper to read OPENAI_API_KEY.
`core/prompts.py`	Loads prompt text files from the /prompts directory by relative path.
`core/state.py`	Defines the shared **PlanState** TypedDict that all LangGraph nodes read/write.
`core/workflow.py`	Builds the LangGraph **StateGraph**: wires nodes, adds the conditional stop path after INTENT_GUARD, and optionally appends RAG_RETRIEVE + EXPLAIN_PLAN when USE_LLM_EXPLAINER=true.

core/nodes

File	What it does
`core/nodes/__init__.py`	Marks **core.nodes** as a package (empty).
`core/nodes/intent_guard.py`	Validates that the request looks like a print-plan input; sets state['stop'] and a friendly message when invalid.
`core/nodes/stl_analyze.py`	Runs STL geometry checks via analyze_stl() and stores state['stl_features']; also back-fills height/width for downstream nodes.
`core/nodes/model_overview.py`	Creates a beginner-friendly one-liner describing what the model *seems* to be using description keywords + STL signals.
`core/nodes/normalize_input.py`	Cleans and validates raw inputs into state['input_norm'], and records assumptions/warnings.

`core/nodes/select_material.py`	Rule-based filament recommendation (PLA/PETG/ABS/ASA/TPU) using description keywords + size hints.
`core/nodes/plan_orientation.py`	Rule-based orientation suggestion using STL bbox/footprint/aspect when available; falls back to user dimensions.
`core/nodes/generate_slicer_settings.py`	Generates slicer knobs (layer height, walls, infill, supports, brim) using material + description hints + STL contact/overhang signals.
`core/nodes/analyze_risks.py`	Detects print risks (mesh issues, adhesion, stability, supports, material-specific risks) and outputs structured risks + mitigations.
`core/nodes/rag_retrieve.py`	Queries the local Chroma vector store for relevant knowledge snippets and stores them in state['rag_context'].
`core/nodes/explain_plan_llm.py`	Uses an LLM to turn the structured plan JSON + warnings/risks (+ optional RAG context) into a readable maker-style explanation; appends beginner model checks.

core/rag

File	What it does
`core/rag/__init__.py`	Marks **core.rag** as a package (empty).
`core/rag/index.py`	Builds/updates the persistent Chroma index from Markdown files in /knowledge using OpenAI embeddings.
`core/rag/retriever.py`	Loads the Chroma vector store from ./.chroma and performs similarity search for the top-k relevant chunks.

core/stl

File	What it does
`core/stl/__init__.py`	Exports analyze_stl as the STL analysis public API.
`core/stl/analyze.py`	STL analysis engine: computes bbox, contact area/ratio, overhang heuristics, watertight/volume flags, and mesh integrity diagnostics.

knowledge

File	What it does
`knowledge/3d_printing_knowledge_base.md`	Small Markdown knowledge base of practical FDM rules (materials, supports, adhesion, infill, etc.) used for RAG grounding.

prompts

File	What it does
`prompts/__init__.py`	Marks **prompts** as a package (empty).
`prompts/system/__init__.py`	Marks **prompts.system** as a package (empty).
`prompts/system/base_system.txt`	System prompt for the explainer LLM (tone + rules).
`prompts/templates/chat_qa.txt`	Template for a chat-style response with optional knowledge context.
`prompts/templates/explain_plan.txt`	Template used by EXPLAIN_PLAN_LLM to format plan JSON + warnings/risks + RAG context into an LLM request.

scripts

File	What it does
`scripts/__init__.py`	Marks **scripts** as a package (empty).
`scripts/build_index.py`	One-shot script to build the Chroma RAG index from the Markdown knowledge base.
`scripts/stl_analyze.py`	CLI helper that prints the STL features dictionary for a given STL file.
`scripts/stl_sanity.py`	Minimal STL sanity script using trimesh (extents/volume/area).

slicebuddy

File	What it does
`slicebuddy/__init__.py`	Marks **slicebuddy** as a Python package (empty).
`slicebuddy/__main__.py`	Allows python -m slicebuddy to run the CLI entry point.
`slicebuddy/cli.py`	Command-line interface: runs the workflow on an STL + use-case string and prints a beginner-friendly report.

root

File	What it does
`requirements.txt`	Pinned Python dependencies for the backend (FastAPI, LangGraph, LangChain, Chroma, trimesh, etc.).
`README.md`	Project overview and usage instructions.
`.env.example`	Example environment variables file (API keys + feature toggles).

ui

File	What it does
`ui/`	Next.js frontend (not detailed here) that calls the FastAPI /plan endpoint; should not commit node_modules or .next to git.