🎯 ASTRAL AI API — Final Architecture Summary

> ***Complete system overview: principles, architecture, components, and workflows***

📋 Table of Contents

Introduction

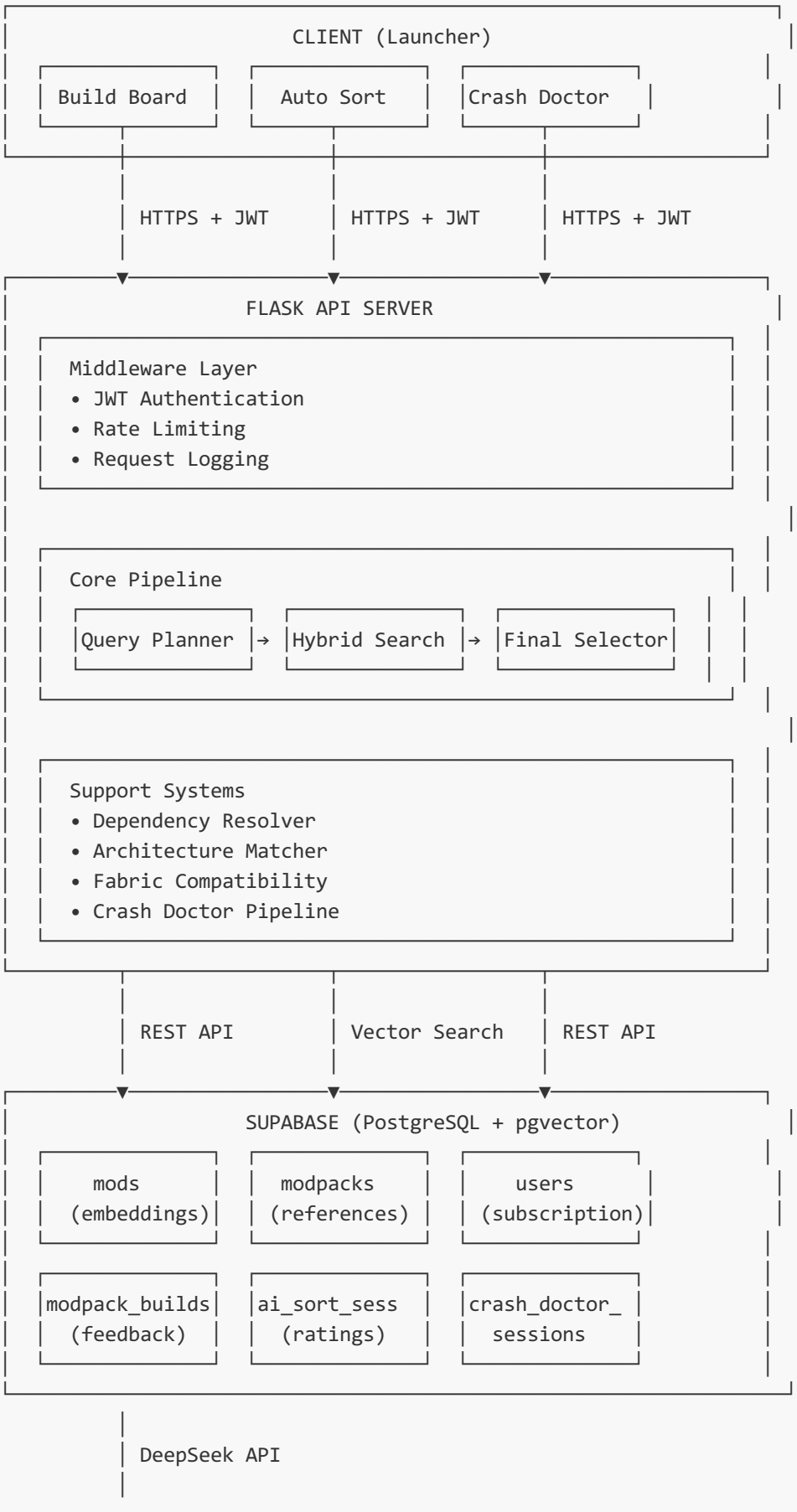
ASTRAL AI API is an intelligent system for automatic Minecraft modpack assembly using advanced AI and RAG (Retrieval-Augmented Generation) technologies. The system analyzes user requests, finds suitable mods through semantic search, and creates ready-to-use modpack configurations.

Key Features

- 🧠 **Conditional Architecture** — system automatically selects optimal processing path
- 🔍 **Hybrid Search** — combination of vector and keyword search
- 🎯 **Capability-Based Search** — semantic mod classification
- 🩺 **Crash Doctor** — automatic crash diagnosis and fixes
- 🔐 **Multi-Layer Protection** — JWT, rate limiting, subscription tiers
- 📊 **Full Transparency** — tracking of all stages and costs

High-Level Architecture

System Overview



```
┌──────────▼──────────────────────────────────────────────────┐
│                    DEEPSEEK AI                              │
│  • Query Planning                                           │
│  • Architecture Planning                                    │
│  • Mod Selection                                            │
│  • Crash Analysis                                           │
└─────────────────────────────────────────────────────────────┘
```

Core System Components

1. Query Planner

Purpose: Analyzes user request and determines optimal processing path.

What it does:

- Determines request type: `simple_add`, `performance`, `themed_pack`
- Creates search plan with priorities
- Decides if Architecture Matcher is needed
- Optimizes queries for search

Input: User request text
Output: JSON search plan with metadata

```
# Example Query Planner output
{
  "request_type": "themed_pack",
  "use_architecture_matcher": true,
  "search_plan": {
    "capabilities_focus": ["combat.weapons", "building.medieval"],
    "baseline_mods": [...],
    "search_queries": [...]
  }
}
```

---

2. Hybrid Search

Purpose: Finds relevant mods using a combination of search methods.

Search Methods:

A. Vector Search (Semantic)

- Uses embeddings (sentence-transformers)

- Understands query semantics
- Finds conceptually similar mods

B. Keyword Search (BM25)

- Exact term matching
- Fast and efficient
- Ideal for specific mod names

C. Reciprocal Rank Fusion

- Combines results from both methods
- Balances relevance
- Reduces bias from single method

Input: Search plan from Query Planner
Output: Ranked candidate list (80-150 mods)

---

3. Architecture Matcher

Purpose: Finds similar reference modpacks for pattern learning.

What it does:

- Searches database of 1000+ analyzed modpacks
- Uses vector search by theme
- Extracts capability patterns
- Identifies baseline mods

When used: Only for themed modpacks (50+ mods)

Output: Top-10 reference modpacks with their architectures

---

4. Architecture Planner

Purpose: Plans category structure for themed modpack.

What it does:

- Analyzes reference modpacks
- Creates 5-12 thematic categories
- Assigns required capabilities to each category
- Estimates mod count per category

Example output:

```
{
  "categories": [
    {
      "name": "Knight's Armory",
      "required_capabilities": ["combat.weapons", "combat.armor"],
      "target_mods": 12
    },
    {
      "name": "Castle Building",
      "required_capabilities": ["building.medieval"],
      "target_mods": 15
    }
  ]
}
```

---

5. Final Selector

Purpose: Intelligent selection of final mods from candidates.

Process:

1. **Local Pre-filtering** (100 → 50 mods)
    - Filter by capabilities
    - Limit by categories (max 6 mods per category)
    - Remove duplicates
2. **AI Selection** (DeepSeek)
    - Analyzes each mod
    - Considers architecture (if available)
    - Ensures category balance
    - Generates explanations

Input: 50-100 candidates
Output: 20-100 selected mods with categories

---

6. Dependency Resolver

Purpose: Automatically adds all required dependencies.

What it does:

- Gets dependency information from database
- Recursively resolves all `required` dependencies
- Checks loader compatibility

- Handles version requirements
- Checks incompatibilities (bidirectional)

Important: No limit on dependency count — all required dependencies are added automatically.

Optimization: Batch database queries (1 query instead of 50)

---

7. Architecture Refiner

Purpose: Adjusts categories based on actually selected mods.

What it does:

- Analyzes real mods after selection
- Splits overloaded categories (15+ mods)
- Merges small categories (1-3 mods)
- Separates libraries from gameplay mods
- Uses thematic names

Example:

```
BEFORE:
- Combat (23 mods) ← overloaded

AFTER:
- Knight's Armory (10 mods)
- Combat Mechanics (8 mods)
- Battle Skills (5 mods)
```

---

8. Smart Categorizer

Purpose: Automatic categorization for simple requests.

When used: For Flow B (simple requests without architecture)

What it does:

- Analyzes description and tags of each mod
- Groups into standard categories
- Simpler than Architecture Refiner (no splitting/merging)

Categories: Performance, Graphics, Utility, World, Gameplay, etc.

---

9. Fabric Compatibility Manager

Purpose: Automatically handles cross-loader compatibility.

What it does:

- Detects when bridges between loaders are needed
- Automatically adds Connector (Fabric ↔ Forge)
- Adds Forgified Fabric API when needed
- Checks version compatibility

Triggers:

- Forge/NeoForge mods detected in Fabric modpack
- `fabric_compat_mode` enabled

---

10. Performance Optimizer

Purpose: Recommends optimization mods for specific loader.

What it does:

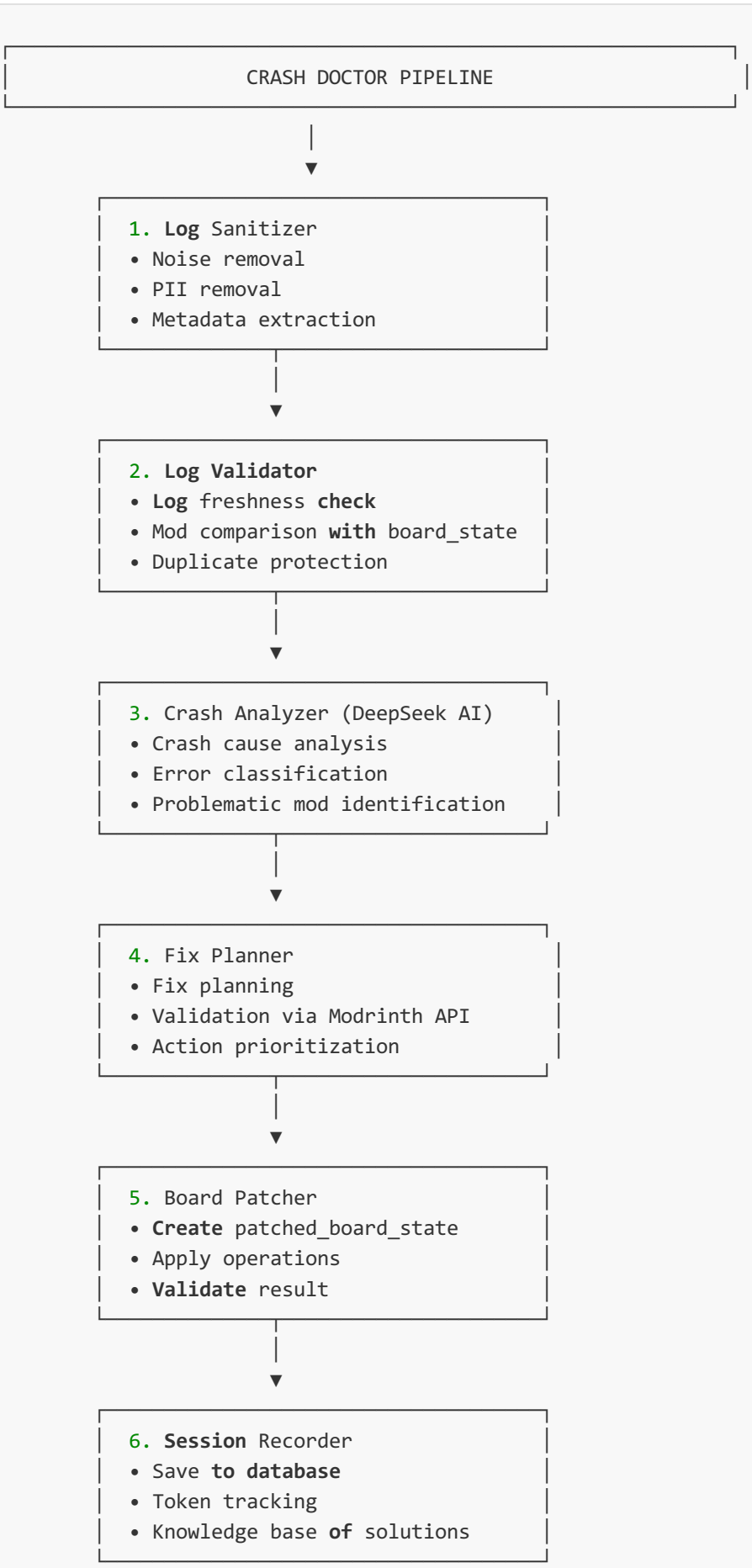
- Knows loader equivalents (Sodium → Embeddium on Forge)
- Considers version-specific features
- Checks optimization coverage (rendering, memory, culling)

Crash Doctor — Crash Diagnostics System

Overview

Crash Doctor is a premium feature for automatic Minecraft crash diagnosis and fixes. The system analyzes crash logs and game logs, determines problem causes, and suggests specific fixes.

Crash Doctor Architecture


CRASH DOCTOR PIPELINE
1. Log Sanitizer
• Noise removal
• PII removal
• Metadata extraction
2. Log Validator
• Log freshness check
• Mod comparison with board_state
• Duplicate protection
3. Crash Analyzer (DeepSeek AI)
• Crash cause analysis
• Error classification
• Problematic mod identification
4. Fix Planner
• Fix planning
• Validation via Modrinth API
• Action prioritization
5. Board Patcher
• Create patched_board_state
• Apply operations
• Validate result
6. Session Recorder
• Save to database
• Token tracking
• Knowledge base of solutions

Crash Doctor Components

1. Log Sanitizer (`log_sanitizer.py`)

Purpose: Cleans and normalizes logs before analysis.

What it does:

- Removes duplicate lines
- Truncates huge data blocks
- Removes PII (file paths, usernames)
- Extracts metadata (MC version, loader, mods)

Output: Cleaned log + metadata

---

2. Log Validator (`log_validator.py`)

Purpose: Checks crash log freshness relative to current board_state.

What it does:

- Extracts mods from crash log
- Compares with mods in board_state
- Calculates match percentage
- Warns if log is outdated (<30% match)

Duplicate Protection:

- Uses MD5 hash of logs
- 1-hour cache per user
- Prevents re-analysis of identical logs

---

3. Crash Analyzer (`crash_analyzer.py`)

Purpose: Analyzes crash log using DeepSeek AI.

What it does:

- Reads entire crash log carefully
- Identifies ALL problem types
- Classifies errors:
  - `mod_conflict` — mod conflict
  - `missing_dependency` — missing dependency
  - `outdated_mod` — outdated mod

- `mixin_error` — mixin transformation error
- `class_not_found` — missing class
- `fabric_mod_on_neoforge` — Fabric mod on NeoForge
- `memory` — memory issues

Output: Structured analysis with problematic mods and causes

---

4. Fix Planner (`fix_planner.py`)

Purpose: Plans specific actions for fixes.

Available actions:

- `remove_mod` — remove problematic mod
- `add_mod` — add missing dependency
- `disable_mod` — disable mod
- `update_mod` — update outdated mod
- `clear_connector_cache` — clear Connector cache

Validation:

- Checks mod existence via Modrinth API
- Checks update availability
- Validates dependency versions

Output: Fix plan with priorities and justifications

---

5. Board Patcher (`board_patcher.py`)

Purpose: Creates fixed version of board_state.

What it does:

- Applies operations from fix plan
- Removes/adds/updates mods
- Preserves board_state structure
- Validates result

Important: Does not apply changes automatically — only creates draft

---

6. Session Recorder (`crash_doctor_recorder.py`)

Purpose: Saves each analysis session to database.

What is saved:

- Unique session ID
- Processed crash log
- Analysis date
- AI solutions
- Tokens used
- Result (suggestions, patched_board_state)

Goal: Building knowledge base of solutions to improve system

SSE Streaming

Crash Doctor uses Server-Sent Events for progress streaming:

Events:

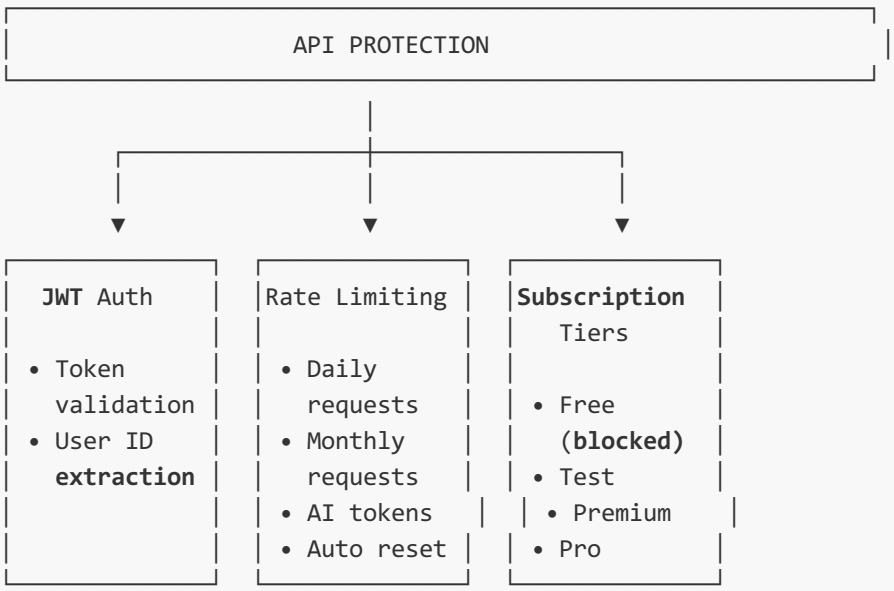
- `progress` — processing stage (validation, sanitization, analysis, planning, patching)
- `complete` — completion with results
- `error` — error with description

Advantages:

- User sees progress in real-time
- Long operations don't appear frozen
- Can cancel operation if needed

Security and Limits

Multi-Layer Protection



1. JWT Authentication

Process:

1. Client sends token in `Authorization: Bearer <token>` header
2. Server validates token via Supabase Auth API
3. Extracts `user_id` from token
4. Checks user existence in database

Important: Token is validated BEFORE all operations

---

2. Rate Limiting

Limits by tier:

Tier	Daily Requests	Monthly Requests	Max Mods/Request	AI Tokens/Month
Free	0	0	0	0
Test	50	1,000	50	100,000
Premium	200	5,000	100	500,000
Pro	Unlimited	Unlimited	200	Unlimited

How it works:

- Check BEFORE request execution

- Counters in database (`users` table)
- Automatic reset:
  - Daily: every day at 00:00 UTC
  - Monthly: 1st of each month

Errors:

- `429 Too Many Requests` — limit exceeded
- `403 Forbidden` — Free tier blocked

---

3. Subscription Tiers

Subscription check:

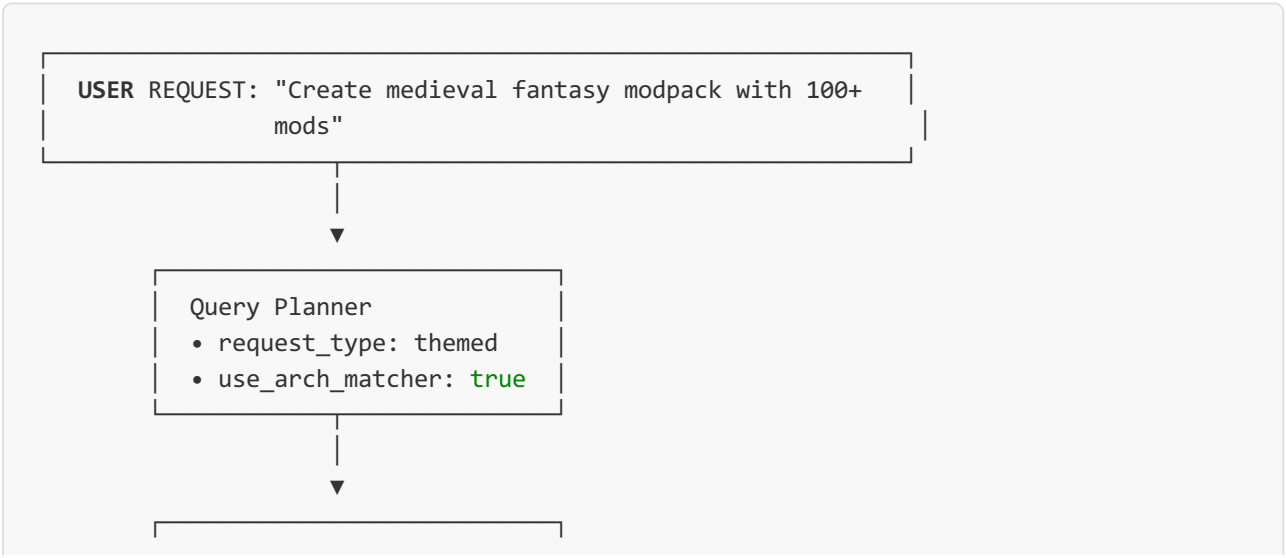
- `subscription_tier` taken ONLY from database (not from client!)
- Free users completely blocked
- Check happens via `@require_subscription` decorator

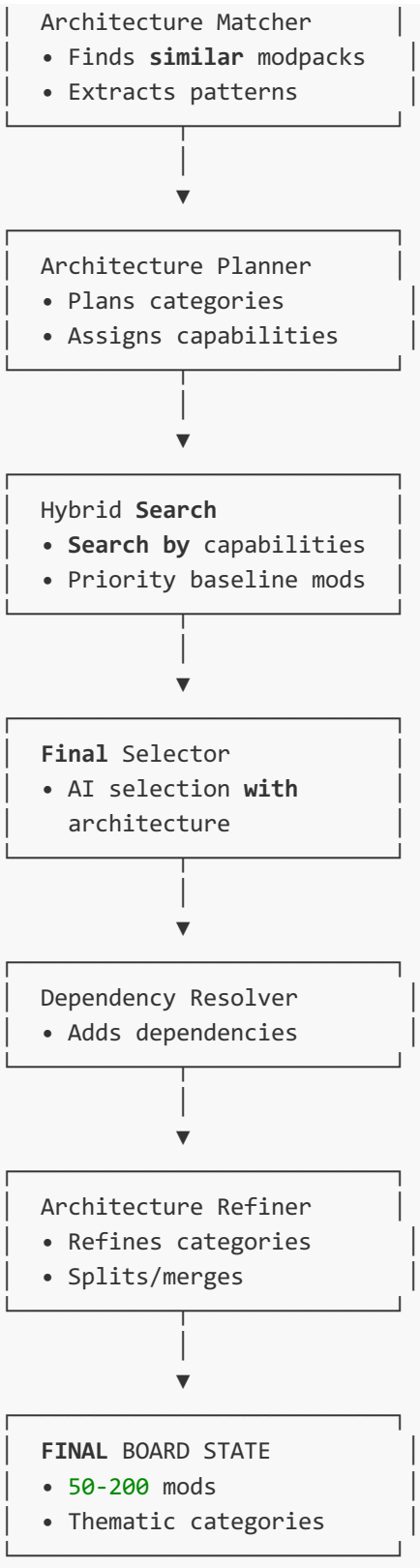
Protected endpoints:

- `/api/ai/build-board` — modpack assembly
- `/api/ai/auto-sort` — automatic sorting
- `/api/ai/crash-doctor/analyze` — crash analysis
- `/api/feedback/ai-sort` — sorting feedback

Visual Flow Diagrams

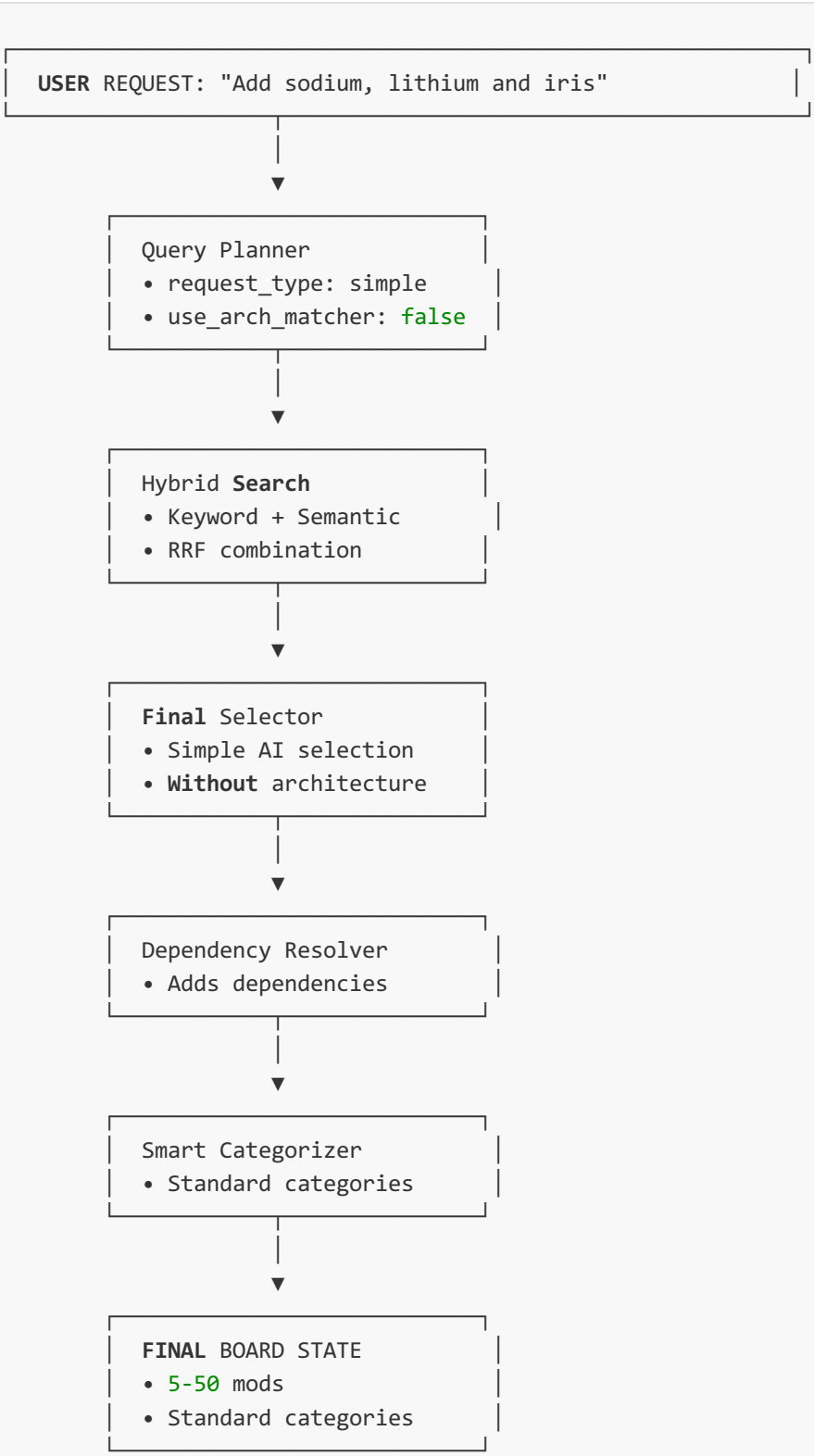
Flow A: Architecture-First (Themed Modpacks)





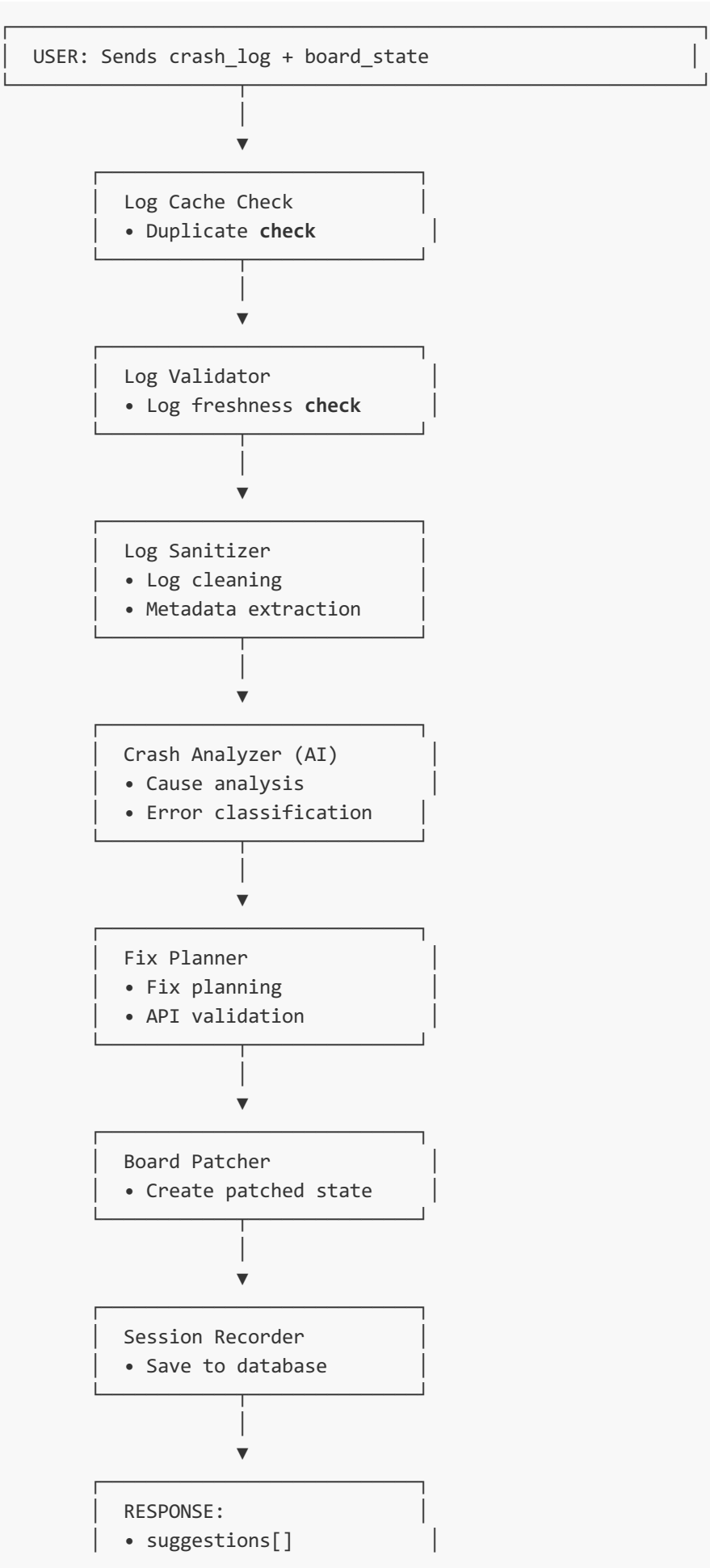
Execution time: ~10-15 seconds
Cost: ~$0.013-$0.026

Flow B: Classic (Simple Requests)



Execution time: ~3-5 seconds
Cost: ~$0.006-$0.013

Crash Doctor Flow


USER: Sends crash_log + board_state
Log Cache Check
• Duplicate check
Log Validator
• Log freshness check
Log Sanitizer
• Log cleaning
• Metadata extraction
Crash Analyzer (AI)
• Cause analysis
• Error classification
Fix Planner
• Fix planning
• API validation
Board Patcher
• Create patched state
Session Recorder
• Save to database
RESPONSE:
• suggestions[]

```
        │  • patched_board_state      │
        │  • confidence               │
        └─────────────────────────────┘
```

Execution time: ~15-30 seconds
Cost: ~$0.02-$0.05 (depends on log size)

Technology Stack

Backend

- **Flask** — REST API framework
- **Python 3.11+** — main language
- **DeepSeek R1** — LLM for reasoning and planning
- **sentence-transformers** — local embeddings (all-MiniLM-L6-v2)
- **requests** — HTTP client for external APIs

Database

- **Supabase** — PostgreSQL + pgvector
- **pgvector** — vector search
- **PostgreSQL** — relational database

External APIs

- **Modrinth API** — mod, version, dependency information
- **Supabase Auth API** — JWT validation
- **DeepSeek API** — AI requests

Infrastructure

- **Cloudflare Tunnel** — tunneling for production
- **SSE (Server-Sent Events)** — progress streaming

Database

Core Tables

`mods`

Stores information about all mods.

Key fields:

- `id` (UUID) — unique ID
- `source_id` (TEXT) — Modrinth project ID
- `embedding` (VECTOR(384)) — vector representation for search
- `capabilities` (TEXT[]) — AI-generated capabilities
- `tags` (TEXT[]) — custom tags
- `loaders` (TEXT[]) — supported loaders
- `game_versions` (TEXT[]) — supported MC versions
- `dependencies` (JSONB) — dependencies
- `incompatibilities` (JSONB) — incompatibilities

Indexes:

- `idx_embedding` — for vector search (ivfflat)

`modpacks`

Reference modpacks for pattern learning.

Key fields:

- `id` (UUID)
- `title`, `slug`, `description`
- `architecture` (JSONB) — capability and provider structure
- `embedding` (VECTOR(384)) — for semantic search
- `mc_version`, `mod_loader`

Usage: Architecture Matcher uses this table to find similar modpacks

---

`users`

User information and subscriptions.

Key fields:

- `id` (UUID) — matches user_id from JWT
- `subscription_tier` (TEXT) — 'free', 'test', 'premium', 'pro'
- `daily_requests_used` (INTEGER) — requests used today
- `monthly_requests_used` (INTEGER) — requests used this month

- `ai_tokens_used` (INTEGER) — AI tokens used
- `last_request_date` (DATE) — last request date
- `custom_limits` (JSONB) — custom limits for VIP

`modpack_builds`

Saves all AI-generated modpacks for learning.

Key fields:

- `id` (TEXT) — unique ID (format: 0000001)
- `user_id` (UUID) — user
- `prompt` (TEXT) — original request
- `architecture` (JSONB) — generated architecture
- `feedback` (JSONB) — user feedback
- `learning_status` (JSONB) — feedback processing status

Usage: System learning based on successful builds

`ai_sort_sessions`

Tracks automatic sorting sessions.

Key fields:

- `id` (TEXT) — unique ID
- `user_id` (UUID)
- `input_mods` (TEXT[]) — mods before sorting
- `categories` (JSONB) — generated categories
- `rating` (INTEGER) — user rating (1-5 stars)
- `created_at` (TIMESTAMPTZ)

Usage: Improving categorization quality

`crash_doctor_sessions`

Crash Doctor solution knowledge base.

Key fields:

- `id` (TEXT) — unique session ID
- `user_id` (UUID)

- `crash_log` (TEXT) — processed crash log
- `board_state` (JSONB) — board state at analysis time
- `suggestions` (JSONB) — suggested fixes
- `patched_board_state` (JSONB) — fixed state
- `confidence` (FLOAT) — solution confidence
- `token_usage` (JSONB) — tokens used
- `created_at` (TIMESTAMPTZ)

Usage: Building knowledge base to improve diagnostics

Development History

Version 1.0 (Legacy)

- Basic AI mod search
- Simple categorization
- Minimal validation

Version 2.0

- Added Hybrid Search
- Improved categorization
- Added dependency support

Version 3.0 (Current) — Conditional Architecture

- **Conditional Architecture** — automatic flow selection
- **Architecture Matcher** — reference modpack search
- **Architecture Planner** — category planning
- **Architecture Refiner** — category refinement
- **Capability-based matching** — capability search
- **Baseline mods** — automatic inclusion of base mods

Crash Doctor (Added later)

- Full crash analysis pipeline
- SSE streaming for progress
- Log validation
- Duplicate caching
- Solution knowledge base

Security (Continuously improved)

- JWT authentication
- Rate limiting by tier
- Subscription tier protection
- Request logging

Key Design Principles

1. Conditional Architecture

Problem: One pipeline doesn't work well for all request types.

Solution: Automatic selection of optimal path based on request analysis.

Advantages:

- ⚡ Fast for simple requests
- 🎯 High quality for complex requests
- 💰 Cost efficiency

---

2. Capabilities over Tags

Problem: Tags are too broad and don't allow precise matching.

Solution: AI-generated capabilities with hierarchical structure.

Advantages:

- More granular classification (50+ capabilities vs 20-30 tags)
- Hierarchical structure (combat.weapons.melee)
- Precise matching for categorization

---

3. Reference Modpack Learning

Idea: Popular modpacks represent proven combinations.

Approach:

1. Parsing 1000+ successful modpacks from Modrinth
2. Extracting capability patterns
3. Using as templates for new builds

Advantages:

- Learning from community experience
- Avoiding common incompatibilities
- Ensuring base quality

---

4. Pipeline Transparency

Principle: Full traceability of all operations.

What is tracked:

- AI token usage
- API request costs
- Search and ranking scores
- Mod selection justifications
- Execution time for each stage

Advantages:

- Debugging and optimization
- Cost monitoring
- Quality analysis

Metrics and Performance

Execution Time

- **Simple requests** (Flow B): 3-5 seconds
- **Themed modpacks** (Flow A): 10-15 seconds
- **Crash Doctor**: 15-30 seconds

Cost (DeepSeek API)

- **Simple requests**: ~$0.006-$0.013
- **Themed modpacks**: ~$0.013-$0.026
- **Crash Doctor**: ~$0.02-$0.05

Optimizations

- **Batch DB queries** — 80% reduction in dependency resolution time
- **Local prefiltering** — reduces AI candidates from 100 to 50

- **Capability-based matching** — more accurate search
- **Log caching** — prevents duplicate analysis

Future Improvements

Planned Features

1. **Semantic Clusters**
    - Pre-built "skeletons" for common modpack types
    - Fast assembly for known themes
2. **Co-occurrence Graph**
    - "Mods that work well together"
    - Learning from successful modpack combinations
3. **User Preference Learning**
    - Tracking user build history
    - Personalized recommendations
4. **Multi-stage Architecture Planning**
    - Iterative refinement with user feedback
    - Interactive category configuration

Conclusion

ASTRAL AI API is a comprehensive system for intelligent Minecraft modpack assembly, combining:

- 🧠 **Advanced AI technologies** (DeepSeek R1)
- 🔍 **Hybrid search** (vector + keyword)
- 🎯 **Semantic understanding** (capabilities)
- 🩺 **Automatic diagnostics** (Crash Doctor)
- 🔐 **Multi-layer protection** (JWT + Rate Limiting)
- 📊 **Full transparency** (tracking all operations)

The system continuously evolves, learning from user feedback and improving recommendation quality.

Document Version: 1.0
Last Updated: 2025-11-12
Author: ASTRAL Team

Built with ❤ for the Minecraft modding community