

Architektur eines modularen Discord-Bots

Ziele und Prinzipien

- **Klare Trennung von Bot und Scanner:** Der Discord-Bot kümmert sich ausschließlich um Discord-spezifische Logik (Events, Commands, Persistenz), während die Bildanalyse durch einen externen **Scanner**-Service erfolgt. Die beiden Komponenten kommunizieren nur über definierte Schnittstellen (HTTP-Aufrufe) ¹ ². Dadurch wird der Bot als Frontend betrachtet, das Anfragen an einen separaten Backend-Dienst stellt, anstatt sämtliche Geschäftslogik in den Bot zu mischen ².
- **Modularität mit Versionierung:** Jede Funktionalität des Bots wird als **Modul** implementiert, und Module tragen einen Versionssuffix (z. B. `_v1`). Wird ein Modul erweitert oder grundlegend überarbeitet, erfolgt dies in einer neuen Version (`_v2`, `_v3` etc.), ohne den alten Code sofort löschen zu müssen ³. Dies fördert langfristige Erweiterbarkeit: Neue Features können hinzugefügt werden, ohne bestehende Funktionen zu brechen ⁴. Die modulare Architektur erleichtert es zudem, Funktionen bei Bedarf ein- oder auszuschalten ⁵.
- **Eindeutige Verantwortlichkeiten & Schichten:** Die Projektstruktur trennt klar zwischen **Discord-Interaktion** und **Kerngeschäftslogik** ⁶. Beispielsweise enthält `index.js` nur das Bootstrapping (Starten des Bots und Laden der Module), in `commands/` liegt ausschließlich die Befehlslogik, in `events/` die Event-Handler für Discord-Ereignisse, und in `lib/` die wiederverwendbaren Kern-Komponenten (z. B. Konfiguration, Scanner-Client, Logging, Datenhaltung) ⁶. Diese Trennung stellt sicher, dass keine Geschäftslogik direkt in den Discord-Ereignishandlern steht und umgekehrt – so bleibt der Code verständlich, testbar und erweiterbar.

(Zusammenfassend gewährleistet diese Architektur, dass neue Features oder Änderungen isoliert entwickelt und integriert werden können. Eine modulare Struktur erleichtert Wartung und Fehlerbehebung, da ein Fehler in einem Modul nicht den gesamten Bot lahmlegt ⁷.)

Projektstruktur

Die Dateien und Ordner sind logisch nach Funktion gruppiert. Ein mögliches Gerüst sieht folgendermaßen aus ⁸ ⁹:

- `index.js` – Hauptprogramm: Startet den Discord-Client und lädt Module.
 - `commands/` – Befehle des Bots (jedes **Command** in einer eigenen Datei):
 - `scanimage_v1.js` – z. B. Befehl zum Scannen eines Bildes (Aufruf etwa via Slash-Command `/scan`).
 - `setscan_v1.js` – z. B. Befehl zum Ein-/Ausschalten der automatischen Bild-Überprüfung.
 - (Weitere Command-Module, ggf. `textscan_v2.js` für zukünftige Textanalyse, etc.)
 - `events/` – Discord-Eventhandler (pro Event-Typ eine Datei):
 - `ready_v1.js` – Handler für das Start-Event des Bots (einmalig bei Login) ¹⁰.
 - `messageCreate_v1.js` – Handler für neue Nachrichten (verarbeitet Befehle *und* checkt Bilder)
- ¹¹.

- `messageReactionAdd_v1.js` – Handler für Reaktionen (z. B. Moderationsreaktionen auf gemeldete Inhalte) ¹².
- *usw.* (weitere Events nach Bedarf, z. B. `guildMemberAdd`, `error` Logging etc.)
- `lib/` – Interne Bibliothek mit wiederverwendbaren Services und Logik:
- `scannerClient_v1.js` – HTTP-Client für den externen Scanner (Bildanalyse) ⁹.
- `botConfig_v1.js` – Modul für Konfiguration laden/speichern ⁹.
- `logger_v1.js` – zentrales Logging-Modul.
- `permissions_v1.js` – ggf. Berechtigungsprüfung für Commands.
- `flaggedStore_v1.js` – (optional) Verwaltung gemeldeter/zu prüfender Inhalte.
- `textAnalyzer_v1.js` – (Platzhalter für zukünftige Text-Analyse Logik).
- *usw.* (z. B. `moderationService_v3.js` in Zukunft für Moderationsfunktionen).
- `config/` – Konfigurationsdateien:
- `bot-config.json` – zentrale Bot-Konfiguration (siehe unten).
- `logs/` – Log-Dateien:
- `bot.log` – Laufzeit-Log des Bots (wird durch `logger_v1` beschrieben).
- (Weitere Logs, z. B. getrennte Logs pro Modul, falls konfiguriert.)

Diese Struktur entspricht bewährten Praktiken für Discord-Bots ¹³. Insbesondere sind Commands, Events und interne Dienste sauber separiert, was Übersichtlichkeit schafft und parallele Entwicklung an unterschiedlichen Modulen erleichtert.

Hinweis: Falls bereits Alt-Code existiert, kann dieser zunächst parallel bestehen bleiben und schrittweise in die neue Struktur migriert werden ¹⁴. Über die Konfiguration kann gesteuert werden, welche Modul-Version aktuell aktiv ist (siehe „versions“ unten), um einen sanften Übergang zu ermöglichen.

Datenmodelle und Konfiguration

Bot-Konfiguration (`bot-config.json`): Dieses JSON-File hält alle einstellbaren Parameter des Bots. Ein vereinfachtes Beispiel:

```
json
{
  "bot": {
    "discordToken": "DISCORD_BOT_TOKEN",
    "ownerIds": ["OWNER_ID_1", "OWNER_ID_2"]
  },
  "guilds": {
    "123456789012345678": {
      "moderatorChannelIds": ["1111", "2222"],
      "rulesChannelId": "3333",
      "moderatorRoleIds": ["4444"],
      "scanEnabled": true
    }
  },
  "scanner": {
    "url": "http://localhost:8000",
    "email": "BOT"
  },
  "paths": {
```

```

    "eventFiles": "./event_files",
    "deletedFiles": "./deleted",
    "logs": "./logs"
  },
  "versions": {
    "events": "v1",
    "commands": "v1",
    "scannerClient": "v1",
    "eventStore": "v1"
  }
}

```

Erläuterung: - `bot`: Globale Bot-Einstellungen (z.B. der Discord-Token und Bot-Owner IDs). Sensible Daten wie Tokens oder API-Schlüssel sollten *nicht* direkt im Code stehen, sondern über die Config/Umgebungsvariablen geladen werden ¹⁷. - `guilds`: Server-spezifische Einstellungen. Im obigen Beispiel pro Guild-ID die IDs relevanter Channels/Rollen für Moderation, sowie ein Schalter `scanEnabled`, der die automatische Bildprüfung für diesen Server aktiviert/deaktiviert. - `scanner`: Verbindungsdaten für den externen Scanner-Service (Base-URL des API und ggf. ein Kennungs-Parameter wie `email` für Authentifizierung). - `paths`: Falls der Bot Dateien speichert, hier Pfade für verschiedene Dateitypen (z.B. temporär gespeicherte Attachments, Logs, etc.). - `versions`: Welche Version jedes Modul-Typs aktuell verwendet wird. Dies erlaubt es z.B. auf eine neue Modulversion umzuschalten, indem hier der Suffix angepasst wird, sofern parallel mehrere Versionen vorhanden sind ¹⁸.

Command-Struktur: Jeder Befehl wird als eigenständiges Modul in der `commands`-Mappe abgelegt. Alle Command-Dateien folgen einem gemeinsamen Schema und exportieren z.B. ein Objekt mit Meta-Daten und der auszuführenden Funktion ¹⁹:

```

js
module.exports = {
  name: 'start',
  version: 'v1',
  description: 'Startet ein Event',
  usage: '!start <tage> <uploads> <name>',
  permissions: ['MANAGE_GUILD'],
  async execute(message, client, args) {
    // Command-Logik
  }
};

```

- `name` gibt den Befehlsnamen (Trigger) an,
- `version` die Implementierungs-Version,
- `description` und `usage` dienen der Dokumentation/des Hilfe-Texts,
- `permissions` listet erforderliche Berechtigungen (für einen ersten Check),
- `execute(...)` enthält die eigentliche Befehlslogik.

Der Bot lädt bei Start alle Dateien in `commands/` (passend zur konfigurierten Versionsnummer) und registriert die Befehle – z.B. indem er sie in einer Map `client.commands` speichert ²⁰. Bei Ausführung eines Befehls wird dann die entsprechende `execute`-Funktion mit den Parametern (z.B. `message`, `client`, `args`) aufgerufen.

Datenmodell Scanner-Ergebnis: Der externe Bilderkennungsdienst liefert pro Bild ein Ergebnis, das vom `scannerClient` in ein internes Format überführt wird. Typischerweise enthält es: - `risk`: ein Wert oder Kategorie für erkannte Risiko-/NSFW-Einstufung (z.B. numerischer Score 0.0–1.0 oder Labels wie "safe" / "nsfw"), - `tags`: eine Liste erkannter Tags oder Objekte im Bild (z.B. ["Hund", "Outdoor", "Person"]), - optional weitere Felder wie `confidence` oder das ungefilterte `raw` Response-Objekt für Logging/Zwecke.

Dieses Ergebnis kann in einem einfachen JS/JSON-Objekt oder einer Klasse `ScanResult` gekapselt werden. Der Bot nutzt diese Daten, um Entscheidungen zu treffen (siehe Ablauf unten). Ähnlich könnte für einen zukünftigen Text-Scanner ein `TextAnalysisResult` definiert werden (etwa mit Feldern für Tonalität, erkannte Schlüsselwörter, etc.).

Weitere Modelle: Intern verwaltet der Bot ggf. Strukturen für temporäre Daten: - Ein **Event-Store** (z.B. `eventStore_v1`) könnte aktive „Events“ oder Wettbewerbe nachverfolgen (wie in obigem Command-Beispiel angedeutet). - Ein **Flagged-Store** (`flaggedStore_v1`) kann genutzt werden, um Beiträge zu speichern, die vom Scanner als kritisch eingestuft wurden und auf Moderationsaktion warten. Dieses Modul würde z.B. die Message-ID, den Nutzer und das Scan-Ergebnis speichern, bis ein Moderator reagiert.

Alle diese Datenmodelle sind gekapselt innerhalb der Module in `lib/` und werden über definierte Schnittstellen von den Commands/Events aus angesprochen, anstatt frei verteilt im Code.

Zentrale Komponenten

Scanner-Client (Bildscanner)

Der **Scanner-Client** (`scannerClient_v1.js`) ist das Bindeglied zwischen dem Bot und dem externen Bildscan-Service. Er kapselt die HTTP-Kommunikation und stellt methodische Aufrufe zur Verfügung, z.B.:

- `scannerClient_v1.ensureToken()` – Stellt sicher, dass ein gültiges Auth-Token für den Scanner vorliegt (ruft ggf. den `/token` Endpunkt des Services auf und cached das Token) ²¹ ²².
- `scannerClient_v1.scanImage(buffer, filename, mimeType)` – Sendet ein Bild (als Binärdaten oder via URL) an den Scanner (z.B. HTTP POST auf `/check`) ²¹. Rückgabewert ist ein strukturierter Scan-Befund (Risiko-Wert, Tags, etc.).
- `scannerClient_v1.scanBatch(buffer, mimeType)` – (Optional) Für spezielle Fälle wie animierte GIFs/Videos: sendet die Datei an `/batch` und erhält aggregierte Ergebnisse für mehrere Frames ²³.
- `scannerClient_v1.getStats()` – (Optional) Ruft Statistiken vom Scanner-Service ab (z.B. Anzahl verarbeiteter Bilder) ²⁴.

Der Scanner-Client liest die nötigen URLs und Zugangsdaten aus der Konfiguration (`config.scanner`) und fügt erforderliche Header (z.B. Auth-Token) hinzu. **Fehlerbehandlung:** Ist kein Token vorhanden oder ist es abgelaufen, versucht `ensureToken()` ein neues Token zu holen. Sollte der Scanner eine **HTTP 403 (Unauthorized)** zurückgeben, so unternimmt der Client einen automatischen Token-Refresh (`/token?email=BOT&renew`) und wiederholt die Anfrage einmal ²². Bei anderen Netzwerkfehlern oder Unerreichbarkeit des Dienstes wirft der Client keinen ungefangenen Fehler, sondern gibt einen Fehlerindikator zurück (etwa `null` oder eine spezielle Resultat-Struktur mit

Fehlercode) – so kann der aufrufende Code entscheiden, wie darauf zu reagieren ist ²². Dieses resiliente Verhalten verhindert, dass ein externer Ausfall den Bot abstürzen lässt.

Event Handling (Discord-Ereignisse)

Alle wichtigen Discord-Events werden in separaten Modulen unter `events/` behandelt. Beim Start registriert der Bot diese Handler beim Discord-Client (z.B. via `client.on('messageCreate', executeFn)`). Wichtige Events und ihre Verantwortung im Kontext dieses Bots:

- `ready` – Wird einmalig beim Start des Bots gefeuert. Der Handler (`ready_v1.js`) loggt z.B. einen erfolgreichen Start (Bot-Tag, Anzahl der Guilds) und führt Sanity-Checks aus ¹⁰. Hier könnte man prüfen, ob alle in der Config erwähnten Server erreichbar sind, oder initiale Daten laden.
- `messageCreate` – Wird bei jeder empfangenen Nachricht aufgerufen. Dieser Handler ist zentral für die Bot-Logik:
 - **Befehle erkennen:** Der Handler prüft zunächst, ob die Nachricht ein Bot-Command ist (z.B. indem sie mit dem definierten Prefix `!` oder einem Slash-Command-Aufruf beginnt). Ist dies der Fall, wird der Befehl samt Argumenten geparkt und die entsprechende Command-Module aus `client.commands` aufgerufen ²⁵. Fehler bei der Ausführung eines Commands werden dabei aufgefangen und geloggt, um den Bot stabil zu halten.
 - **Inhalte scannen:** Wenn es kein Bot-Command ist, prüft der Handler, ob die Nachricht Anhänge oder Bild-Links enthält und ob für den Server die automatische Prüfung aktiviert ist (`scanEnabled` in der Config) ²⁶. Für alle relevanten Attachments (Bilder/Videos) ruft der Bot den oben genannten `scannerClient_v1.scanImage(...)` auf ²⁷. Die zurückgelieferten Ergebnisse (Risiko, Tags) werden ausgewertet:
 - Ist der **Risiko-Wert hoch** (z.B. NSFW erkannt), kann der Bot eine Moderationsaktion vorbereiten: Zum Beispiel den Vorfall in einen der konfigurierten `moderatorChannelIds` melden (Embed mit Bild, User, Risiko-Wert/Tags) und den Inhalt in einem internen `flaggedStore` vormerken. Je nach Konfiguration könnte der Bot den Beitrag auch **automatisch entfernen**, falls `delete` aktiviert ist, und den User per privater Nachricht warnen.
 - Bei **geringem Risiko** oder unbedenklichen Inhalten unternimmt der Bot nichts weiter außer ggf. die Tags zu loggen oder für Statistiken zu zählen.
 - Alle erkannten Tags können für spätere Auswertungen (z.B. welche Inhalte wie häufig gepostet wurden) in einem Statistik-Modul mitgezählt werden.
 - **Event-bezogene Logik:** (Optional) In diesem Bot-Beispiel gibt es „Events“ (Wettbewerbe), bei denen Bilduploads gezählt werden. Der Handler ruft daher – falls der Beitrag in einem aktiven Event-Channel gepostet wurde – zusätzlich `eventStore_v1.registerUpload(...)` auf, um den Upload für das Event zu registrieren (z.B. für eine Punktzahl oder spätere Auswertung) ²⁸.
 - **Fehlerbehandlung:** Etwaige Fehler (z.B. Exception im Scanner-Client oder in Command-Execution) werden *nicht* ungefiltert weitergeworfen, sondern innerhalb des Handlers abgefangen und geloggt ²⁹. So wird verhindert, dass ein einzelner Vorfall den Bot-Prozess abstürzt ⁷. Je nach Fehlerfall kann der Bot den Nutzer informieren (bei einem Command z.B. mit einer Chat-Nachricht „⚠ Fehler bei der Verarbeitung, bitte später erneut versuchen.“) ³⁰, oder still im Hintergrund bleiben (bei einem Scanner-Ausfall während Auto-Scan nur Log-Eintrag für Admins).
- `messageReactionAdd` – Dieser Event-Handler wird aktiv, wenn jemand auf eine Nachricht reagiert (Emoji hinzugefügt). In unserem Kontext dient dies vor allem Moderationszwecken ¹².

Beispielsweise könnte das Modul `messageReactionAdd_v1.js` folgendermaßen genutzt werden:

- Es filtert auf bestimmte Channels (etwa den Moderations-Logchannel) und bestimmte Emojis.
- Bei 🤔 oder 🗑️-Reaktionen: Markiert ein Moderator die gemeldete Nachricht als *gesehen* bzw. *bewertet* (dies könnte z.B. für Statistikzwecke verwendet werden, oder um dem meldenden User Feedback zu geben).
- Bei 🗑️ (rotes Kreuz): Der Moderator signalisiert, dass der Inhalt entfernt werden soll – der Bot löscht daraufhin die Originalnachricht und verschiebt die zugehörige Datei ggf. in einen geschützten Ordner (`./deleted/` in `paths`) ³¹.
- Bei ⚠️ (Warnzeichen): Der Bot sendet automatisiert eine Verwarnung per DM an den Uploader der Nachricht (mit Hinweis auf die Server-Regeln, z.B. Link zum `rulesChannelId`) ³¹.
- Zusätzlich können Reaktionen genutzt werden, um Punkte oder Flags in den entsprechenden Stores zu setzen – z.B. `eventStore_v1.applyReaction(...)` um die Wertung eines Event-Beitrags zu beeinflussen, oder Einträge im `flaggedStore_v1` als bearbeitet zu markieren ³¹.

(Ähnliche Event-Module könnten für andere Ereignisse hinzugefügt werden – etwa `guildMemberAdd` für Willkommensnachrichten, oder ein regelmäßiges Intervallvent für Hintergrundaufgaben. Die Architektur lässt hier viel Spielraum, ohne die bestehenden Module zu beeinflussen.)

Command Handling

Die **Befehlsverarbeitung** ist vom obigen Nachrichtenevent entkoppelt gestaltet. Das Laden und Registrieren der Commands passiert zentral in der Startphase des Bots. Wie erwähnt, werden alle Command-Module aus dem `commands/` Ordner, die dem konfigurierten Versionssuffix entsprechen, dynamisch importiert und z.B. in einer Collection (`client.commands`) abgelegt ²⁰.

Bei Eintreffen einer Nachricht, die als Befehl identifiziert wurde, übernimmt der Message-Event-Handler nur noch das Dispatching: Er liest den Command-Namen und die Parameter aus der Nachricht und ruft `client.commands.get(name).execute(message, client, args)` auf ²⁵. Die eigentliche Logik liegt also vollständig im jeweiligen Command-Modul (siehe Command-Schema oben). Dadurch kann man Commands einfach hinzufügen oder ändern, ohne den zentralen Bot-Code anfassen zu müssen – der Event-Handler bleibt unverändert.

Um die **Überschneidung mit Discord-spezifischen Interaktionen** weiter zu minimieren, könnte man die Commands auch als Discord **Slash Commands** implementieren. In dem Fall würde anstelle eines Prefix-Parsers die Registrierung via Discord-API (z.B. beim Start mit `REST.put` der Commands) erfolgen, und im Event `interactionCreate` würde der Bot dann ähnlich die hinterlegten Command-Handler aufrufen. Die Modularisierung im Code (ein File pro Command) bleibt davon unberührt – es ändert sich nur die Art des Triggers.

Konfigurationsmanagement

Die zentrale Config (`botConfig_v1` in `lib/`) bietet Methoden, um die JSON-Datei zu laden, Änderungen zu speichern und bequem darauf zuzugreifen ³². Beispielhafte Schnittstelle: - `botConfig_v1.loadConfig()` – Liest die `bot-config.json` von der Disk, füllt Defaults ein und gibt ein Config-Objekt zurück ³³. - `botConfig_v1.saveConfig(cfg)` – Schreibt Änderungen zurück in die JSON-Datei, z.B. wenn ein Admin zur Laufzeit Einstellungen ändert (etwa via Command) ³³. - `botConfig_v1.getGuildConfig(cfg, guildId)` – Hilfsfunktion, um sicher auf die Config für einen bestimmten Server zuzugreifen (fällt z.B. zurück auf Defaultwerte, falls der Eintrag fehlt) ³⁴.

Beim Bot-Start wird `loadConfig()` genutzt, um die Einstellungen zu laden. Der Discord-Client (`client`) kann die Config dann als Property erhalten (`client.config`), sodass jeder Teil des Bots darauf zugreifen kann ³⁵. Beispielsweise prüft der Message-Event-Handler `client.config.guilds[guildId].scanEnabled`, um zu entscheiden, ob er den Scanner aufruft.

Wie bereits erwähnt, sollten **sensible Daten** (Tokens, API Keys) möglichst nicht in dieser für das Repository bestimmten JSON im Klartext stehen. Häufig wird ein .env-Mechanismus genutzt: `bot-config.json` enthält nur Platzhalter oder Pfade, und zur Laufzeit ersetzt `botConfig_v1` diese mit echten Werten aus Umgebungsvariablen oder einem nicht eingetragenen Secret-File ¹⁷. Dies erhöht die Sicherheit und verhindert, dass z.B. der Discord-Bot-Token in falsche Hände gerät ³⁶.

Logging und Fehlerbehandlung

Ein **Logging-Modul** (`logger_v1`) sorgt dafür, dass wichtige Ereignisse und Fehler zentral erfasst werden. Bereits im `index.js` wird z.B. ein Handler für unhandled Promise Rejections oder uncaught Exceptions registriert, der solche Fälle abfängt und sauber ins Log schreibt ³⁷. So kann der Bot kontrolliert weiterlaufen oder sich geordnet neu starten, anstatt abrupt zu crashen.

Die Logs (Textdateien im `logs/` Ordner) helfen bei der Fehlersuche und Nachvollziehbarkeit von Bot-Aktionen. Es bietet sich an, pro **Modul** oder zumindest pro Hauptkomponente einen separaten Logger zu verwenden (z.B. Präfix im Logeintrag), um Probleme schneller zu lokalisieren ³⁸. In einem komplexeren Setup könnte auch ein Logging-Framework eingesetzt werden, das z.B. nach Schweregrad filtert (Info/Warn/Error) und Logs rotiert.

Durch die umfassende Fehlerbehandlung auf Modulebene wird sichergestellt, dass ein Ausfall – sei es ein nicht erreichbarer Scanner oder ein Bug in einer Command-Funktion – nicht den gesamten Bot lahmlegt. Jede asynchrone Operation ist von try-catch umgeben oder nutzt `.catch(...)`-Handler. Im Fehlerfall versucht der Bot das Nötigste (Ressourcen freigeben, ggf. Retry bei transienten Fehlern) und loggt den Vorfall. Dem Nutzer wird, falls relevant, eine verständliche Fehlermeldung angezeigt (anstatt eines technischen Stacktraces) ³⁰. Dieses Prinzip *"eine fehlerhafte Anfrage bringt nicht den ganzen Dienst zu Fall"* ist essenziell für einen dauerhaft laufenden Bot ⁷.

Beispiel: Bildscan-Modul (Flow & Fehlerhandling)

Als initiales Modul sei der **Bildscanner (Image Scan)** in Version 1 beschrieben, der eingehende Bilder überprüft (NSFW-Check, Tagging). Der Ablauf gliedert sich in folgende Schritte:

1. **Trigger:** Ein Benutzer postet eine Nachricht mit einem oder mehreren Bildern (Attachments) in einem Channel. Alternativ kann auch explizit per Befehl (z.B. `/scan`) ein Bild-URL eingegeben werden – beide Wege führen dazu, dass der Bot den Scan startet. Der relevante Event-Handler (`messageCreate_v1`) erkennt das Szenario: Entweder es ist ein Command (`/scan`) oder (bei Auto-Scan) eine normale Nachricht mit Attachment ¹¹.
2. **Vorbereitung der Anfrage:** Der Bot sammelt alle zu prüfenden Bild-Dateien. Bei Attachments liest er die URLs oder lädt die Dateien in einen Buffer (je nach API), bei einem Command mit URL wird diese verwendet. Falls nötig, werden große Dateien übersprungen oder abgelehnt (z.B. könnte `scannerClient_v1` eine Größenbeschränkung prüfen, um unnötige Anfragen zu vermeiden).

3. **Aufruf des externen Scanners:** Für jede Bilddatei ruft der Bot `scannerClient_v1.scanImage(buffer, filename, mimeType)` auf ²⁷. Dieser kümmert sich darum, die HTTP-Anfrage an den externen Dienst zu stellen. Der Bot wartet *asynchron* auf die Antwort. Sollte ein Bild ein animiertes Format (GIF/WebM) sein, könnte statt `scanImage` die Methode `scanBatch` verwendet werden, die mehrere Frames analysiert ²³.
4. **Empfang und Auswertung des Ergebnisses:** Der Scanner-Service antwortet typischerweise mit JSON-Daten (siehe Datenmodell oben). Ein Beispiel-Antwort könnte sein: `{ "risk": 0.95, "tags": ["nsfw", "nudity"], "raw": {...} }` für ein ungeeignetes Bild. Der Scanner-Client wandelt das in ein JS-Objekt `result` um. Nun prüft der Bot:
 5. **Risk/NSFW-Level:** Überschreitet `result.risk` einen definierten Schwellenwert (z.B. 0.8), so behandelt der Bot das Bild als **kritisch**. Andernfalls gilt es als **unbedenklich**.
 6. **Tags:** Die im Bild erkannten `result.tags` (z.B. "Waffe", "Blut" für Gewalt, oder "Badeanzug" für möglicherweise heikle Inhalte) werden ggf. mit konfigurierten **Filterregeln** abgeglichen. So kann man bestimmte Tags als immer unerwünscht markieren.
 7. **Beispielentscheidungen:** Ist ein Bild als kritisch eingestuft, kann der Bot es *automatisch löschen* und eine Meldung im Moderationskanal erzeugen (inkl. der Tags und einer Warnung). Alternativ – wie im vorigen Abschnitt skizziert – markiert der Bot die Nachricht zunächst nur (z.B. mit einer -Reaktion von sich selbst) und trägt sie in `flaggedStore_v1` ein, um einem Moderator die finale Entscheidung zu überlassen. Bei unbedenklichen Bildern kann der Bot z.B. nichts weiter tun oder in einem Statistikmodul einen Zähler für gefundene Tags erhöhen.
8. **Rückmeldung/Follow-up:** Abhängig vom Ergebnis unternimmt der Bot nun Aktionen:
 9. Im **Auto-Moderationsmodus:** Entfernt er sofort die Nachricht (bei Verstoß) und benachrichtigt den User: *"Dein Bild wurde entfernt, da es gegen die Regeln verstößt (NSFW erkannt)."* Gleichzeitig wird ein Eintrag im Mod-Log gepostet.
 10. Im **Review-Modus:** Lässt er die Nachricht vorerst stehen, versieht sie aber mit einer Reaktion (⚠) oder einer internen Markierung. Im Mod-Channel erscheint eine Meldung: *"Bild von @User zur Überprüfung markiert (Tags: X, Y)."* Ein Moderator kann dann per Reaktion entscheiden (siehe oben).
 11. Bei **erfolgreichem Scan ohne Befund:** Kann der Bot dem Benutzer optional feedback geben, z.B. *"Bild gescannt, keine Probleme festgestellt."* – wobei in der Praxis Bots meist nur im Fehler- oder Moderationsfall überhaupt etwas sagen, um den Chat nicht vollzuspannen.
 12. **Statistiken:** Unabhängig vom Moderationspfad können die extrahierten Tags und ein Zähler für NSFW-Funde in einer Statistik gesammelt werden (z.B. Anzahl an entfernten Bildern pro Tag, häufigste Tags etc., was später über einen Statistik-Command abrufbar ist).
13. **Fehlerfälle und Retry:** Was passiert, wenn der Scanner nicht antwortet?
14. **Timeout oder Netzwerkfehler:** Der `scannerClient_v1` erkennt dies und gibt einen Fehler zurück (oder wirft eine Exception, die im Handler abgefangen wird). Der Bot loggt den Fehler und teilt dem Benutzer bei einem expliziten Scan-Command mit, dass der Service aktuell nicht verfügbar ist ³⁰. Bei Auto-Scan könnte er still schweigen, nur Admins informieren. Möglicherweise wird ein Retry-Mechanismus implementiert, der nach einigen Sekunden einen zweiten Versuch unternimmt, bevor aufgegeben wird ³⁹.

15. **HTTP-Fehlercode (nicht 200):** Ist z.B. das Auth-Token abgelaufen (403 Forbidden), unternimmt der `scannerClient_v1` automatisch den Token-Refresh und wiederholt die Anfrage einmal ²². Schlägt auch dies fehl, wird der Fehler hochgereicht.
16. **Unerwartetes Response-Format:** Wenn der externe Dienst sich anders verhält als erwartet, fängt der Client eine JSON-Parse-Exception ab. Solche Fälle werden ebenfalls geloggt und führen ggf. zu einer Meldung "Scanning derzeit nicht möglich."
17. In allen Fällen gilt: **kein Absturz** – Fehler werden intern behandelt, der Bot bleibt responsiv. Im schlimmsten Fall wird halt ein bestimmtes Bild nicht geprüft, aber der Bot an sich läuft weiter ⁷.
18. **Logging:** Jedes gescannte Bild – ob Ergebnis positiv oder negativ – wird im Log festgehalten, inkl. relevanter Metadaten (Server, Channel, User, Ergebnis). Im Fehlerfall enthält das Log auch die Fehlermeldung oder HTTP-Status. Diese Transparenz hilft später bei der Feinjustierung (z.B. Schwellenwerte anpassen, wenn zu vieles fälschlich angeschlagen ist) und bei der Nachvollziehbarkeit von Moderationsentscheidungen.

Zusammengefasst zeigt dieses Beispiel, wie das Zusammenspiel von Event-Handler, Command, Scanner-Client und Config funktioniert, um einen Anwendungsfall – die automatisierte Bildinhaltskontrolle – umzusetzen. Die saubere Modultrennung ermöglicht es, jeden Teil (Erkennung, Auswertung, Moderation) getrennt weiterzuentwickeln oder auszutauschen, ohne das Gesamtsystem zu destabilisieren.

Erweiterung und Versionierung neuer Module

Die Architektur ist von Anfang an darauf ausgelegt, neue Module und Funktionen unkompliziert hinzuzufügen. Durch die Versionierung kann dies sogar schrittweise erfolgen, ohne alte Funktionalität sofort entfernen zu müssen ³. Im Folgenden einige Hinweise, wie künftige Module integriert werden können:

- **Textanalyse-Modul (z. B. `TextScanner_v2`):** Angenommen, nach dem Bildscanner soll ein Modul zur **Textanalyse** (z.B. Erkennung von Toxicity, Spam oder Keywords in Nachrichten) hinzugefügt werden. Dieses würde analog aufgebaut:
 - Ein Service in `lib/` (etwa `textAnalyzer_v1.js`), der entweder einen externen NLP-Dienst aufruft oder lokal Bibliotheken nutzt, um Texte zu analysieren.
 - Gegebenenfalls ein Event-Handler (in `events/`), der auf `messageCreate` oder `messageUpdate` lauscht und für jede neue Nachricht `textAnalyzer_v1.analyze(content)` ausführt. Alternativ könnte man die Logik auch im bestehenden `messageCreate`-Handler unterbringen – dank modularem Design ließe sich das aber leicht entkoppeln, indem der Handler einfach beide Module (Bild und Text) nacheinander aufruft.
 - Optionale Commands unter `commands/`, z.B. `/analyzetext` um einen bestimmten Text oder die letzten N Nachrichten manuell prüfen zu lassen.
 - Versionierung: Da es sich um ein neues Feature handelt, beginnt man bei Version `v1`. Sollte später die Analyse verbessert werden (z.B. **TextScanner_v2** mit einem neuen Machine-Learning-Modell), könnte man das Modul als neue Datei `textAnalyzer_v2.js` hinzufügen. In der Config könnte dann für bestimmte Guilds oder global eingestellt werden, welche Version genutzt wird (`config.versions.textAnalyzer: "v2"`), falls ein stufenweiser Rollout gewünscht ist.

- **Zusammenspiel:** Das Textanalyse-Modul kann eigenständig arbeiten, aber auch Ergebnisse an andere Module liefern. Beispielsweise könnte es erkannte beleidigende Sprache an ein Moderationsmodul weitergeben, ähnlich wie der Bildscanner NSFW-Inhalte meldet. Wichtig ist, klare **Schnittstellen** zu definieren – z.B. eine gemeinsame Funktion `Moderation.reportIssue(type, details)`, die von Bild- und Textscanner aufgerufen wird, um Funde an das Moderationssystem zu melden.
- **Statistik-Modul:** Mit zunehmenden Funktionen lohnt sich ein Modul, das **Statistiken** sammelt und auf Anfrage bereitstellt. Dies könnte z.B. `stats_v1.js` in `lib/` sein, mit einer zentralen Sammlung von Zählern (Anzahl gescannter Bilder, erkannte NSFW-Inhalte, häufigste Tags, etc.). Commands wie `/stats` oder `/topTags` könnten darauf zugreifen und dem Admin Auswertungen liefern. Die Implementierung eines Statistikmoduls ist weitgehend unabhängig von Discord – es aggregiert Daten, die aus anderen Modulen (Bild/Textscanner, Moderation) an es gemeldet werden. Dank der modularen Architektur kann man solche Interaktionen gut kapseln (z.B. der Scanner ruft `Stats.recordImageScan(result)` auf). Versionierung spielt hier eine geringere Rolle, es sei denn, die Art der erhobenen Daten ändert sich drastisch (dann würde man analog eine neue Version einführen).
- **Moderations-Modul (z. B. `Moderation_v3`):** Ein Herzstück, das über reine Erkennung hinausgeht, ist das Moderationssystem. In unserem Bot-Beispiel wurden bereits Teile davon in den Reaction-Handletern und Stores skizziert. Ein ausgebautes Moderationsmodul könnte folgende Elemente haben:
 - **Befehlshandling:** Commands wie `/warn @User [Grund]`, `/ban @User`, `/history @User` usw., die Moderatoren nutzen können. Diese würden als eigene Dateien unter `commands/` liegen (z.B. `warn_v1.js`, `ban_v1.js`).
 - **Interne Logik:** Ein `moderation_v1.js` in `lib/` könnte die Geschäftslogik kapseln – z.B. Verwarnungen in einer Datenbank oder Datei speichern, Zählungen führen (für automatische Eskalation bei X Verwarnungen) und Integrationen mit dem Discord-API (Rollen entziehen, kicken/bannen).
 - **Event-Hooks:** Das Moderationsmodul würde die von Scannern bereitgestellten Informationen nutzen. Wenn der Bildscanner etwa einen Verstoß meldet, könnte `Moderation_v3` automatisch eine Verwarnung verbuchen und – falls eingestellt – direkt eingreifen. Auch das Entfernen von Nachrichten aufgrund bestimmter Keywords (Textscanner) könnte hier zentral gesteuert werden.
 - **Versionierung:** Moderationsfeatures wachsen meist organisch. Version 1 könnte simple Verwarnungen bieten, Version 2 fügt vielleicht automatische Temp-Bans hinzu, Version 3 ein komplettes Punktesystem. Durch die Versionierung im Dateinamen kann das Team parallel an einer erweiterten Version arbeiten, ohne die stabile Version 1 zu gefährden. Wenn fertig, wird auf `Moderation_v3` umgestellt. Alte Versionen können für Referenz oder Rollback-Zwecke erhalten bleiben ³.
- **Integration neuer Module:** Um ein neues Modul einzubinden, sind folgende Schritte typisch:
 - **Codestruktur anlegen:** Neues File im passenden Ordner (oder mehrere, falls es Commands + Service umfasst). Namensgebung mit nächster verfügbarer Version.
 - **Exports/Schnittstelle definieren:** Konsistent mit vorhandenen Modulen – z.B. bei Commands das oben beschriebene Schema verwenden; bei Services klare Funktionen exportieren.
 - **Module laden:** Falls es ein neuer Event-Typ ist, muss `index.js` einen Listener dafür registrieren (oder der bestehende Event-Handler ruft das neue Modul auf). Bei neuen

Commands diese wie gehabt dynamisch laden. Ggf. `config.versions` ergänzen, damit das Modul aktiviert wird.

- **Dokumentation/Changelog:** Vermerken, welche neuen Fähigkeiten hinzukamen. Da die Version im Modulnamen steckt, ist nachvollziehbar, welche Iteration vorliegt. Empfehlenswert ist es, in einer Doku festzuhalten, was sich zwischen z.B. `_v2` und `_v3` geändert hat (neue Parameter, anderes Verhalten etc.).
- **Testing:** Dank entkoppelter Logik können neue Module isoliert getestet werden. Z.B. kann man `textAnalyzer_v1.analyze("Test")` offline testen, ohne Discord. Integrationstests im Bot-Kontext prüfen dann das Zusammenspiel (z.B. ob der Message-Event-Handler das Modul tatsächlich aufruft). Eine saubere Modularchitektur fördert auch das Schreiben von Unit-Tests für Kernfunktionen, da diese nicht von Discord-Internas abhängen ².

Zusammengefasst bedeutet **Versionierung** hier nicht nur, den Zustand des Codes zu markieren, sondern aktiv nebeneinander verschiedene Implementierungen verwalten zu können. Dies erleichtert es, größere Refactorings oder Technologie-Upgrades (etwa einen neuen Scanner-Service) vorzubereiten, während der alte noch läuft. Die Config-Schalter unter „versions“ können sogar erlauben, modulweise ein Rollback zu machen, sollte eine neue Version unerwartete Probleme bereiten – man würde einfach wieder `"scannerClient": "v1"` einstellen, falls `"v2"` zickt. Diese Flexibilität ist ein großer Vorteil einer modularen Bot-Architektur ³.

Trennung von Kernlogik und Discord-Interaktionen

Ein zentrales Architekturmuster bei diesem Bot ist die **Schichtung** in eine plattform-unabhängige Kernlogik und eine Discord-spezifische Interaktionsschicht. Die Kernmodule (Scanner-Client, Text-Analyzer, Event/Flag Stores, Moderationslogik etc.) sind so geschrieben, dass sie *keine* Annahmen über Discord treffen – sie könnten theoretisch auch von einem anderen Frontend (einer Web-App, einem Slack-Bot usw.) genutzt werden ². Beispielsweise kennt `scannerClient_v1.scanImage()` nur Bilddaten und liefert Analyseergebnisse zurück, ohne zu wissen, dass die Daten aus Discord stammen. Dadurch ergeben sich mehrere Vorteile:

- **Wiederverwendbarkeit:** Sollte der Service auf einer anderen Plattform angeboten werden (z.B. zusätzlich ein Slack-Bot oder eine Weboberfläche), kann die gesamte Kernlogik wiederverwendet werden. Lediglich die „Frontend“-Schicht müsste neu implementiert werden (andere API-Aufrufe), die Module in `lib/` bleiben unverändert ⁴⁰.
- **Testbarkeit:** Kernfunktionen können in Isolation getestet werden (Unit-Tests), was die Qualität steigert. Discord-spezifische Objekte (wie `message` oder `client`) werden nur in den oberen Schichten verwendet. Im Test kann man sie durch einfache Dummy-Daten ersetzen. So war z.B. das Testen des Gültigkeitsbereichs der Scanner-Funktionen deutlich einfacher möglich.
- **Wartbarkeit:** Änderungen an der Geschäftslogik (z.B. neues Scanner-API, veränderte Moderationsregeln) geschehen in den Kernmodulen, ohne dass man sich mit Discord-Events befassen muss. Umgekehrt kann ein Discord-API-Update (etwa neue Event-Typen, Deprecation alter Methoden) in der Interaktionsschicht angepasst werden, ohne die interne Logik zu tangieren.
- **Klarheit im Code:** Durch die saubere Aufteilung ist sofort ersichtlich, wo bestimmte Logik verortet ist. Ein Entwickler, der die Tag-Erkennung verbessern will, schaut in `scannerClient_v1.js` nach. Jemand, der die Bot-Commands erweitern will, konzentriert sich auf den `commands/`-Ordner. Diese Separation of Concerns macht das Projekt für mehrere Entwickler und langfristig nachvollziehbar.

In der Praxis manifestiert sich diese Trennung z.B. darin, dass `index.js` beim Start die **Brücke** schlägt: Dort werden die Discord-Events mit den internen Funktionen verknüpft, z.B.

`client.on('messageCreate', (...){ events/messageCreate_v1.execute(...); })`, und es werden Instanzen der Service-Klassen erstellt und am Client objekt hinterlegt (z.B. `client.scanner = scannerClient_v1`, wie in der Init-Liste ersichtlich) ³⁵. Nach dem Start laufen die Kernmodule „unter der Haube“ des Discord-Bots und könnten theoretisch ausgetauscht werden, ohne die Bot-Mechanik zu ändern – genau das wird mit dem Versionierungskonzept auch genutzt.

Abschließend lässt sich sagen, dass diese Bot-Architektur durch **Modularität, klare Schnitte und Versionierbarkeit** ausgezeichnet für langfristige Erweiterungen gerüstet ist. Neue Funktionen können als Module hinzugefügt werden, bestehende Komponenten lassen sich in neuen Versionen verbessern, und der Bot bleibt dabei stabil und übersichtlich. Die Trennung von Discord-spezifischer Logik und allgemeiner Kernlogik stellt sicher, dass der Fokus jeder Komponente klar bleibt und die Wiederverwendbarkeit sowie Wartbarkeit maximiert wird. Dieses Fundament ermöglicht es, den Bot schrittweise zu einem umfangreichen System (von Bild- und Textmoderation über Statistiken bis hin zu beliebigen weiteren Features) auszubauen, ohne dass die Struktur degeneriert – im Gegenteil, jede Erweiterung folgt den etablierten Mustern dieser **professionellen Bot-Architektur**. ^{1 4}

^{1 3 6 8 9 10 11 12 14 15 16 18 19 20 21 22 23 24 25 26 27 28 29 31 32 33 34 35 37}

pixai-bot-phase2-bot-only.md

file:///file-XUWu4EP4y1WuMp3AGMjdi6

^{2 40} Architecting discord bot the right way | by Nikhil Taneja | Medium

<https://itsnikhil.medium.com/architecting-discord-bot-the-right-way-46e426a0b995>

^{4 7 36 38} Building a Multi-Purpose Discord Bot: A Developer's Journey | by Balasubramanian Manish | Medium

<https://itzmaniss.medium.com/building-a-multi-purpose-discord-bot-a-developers-journey-62dbbffc40f1>

⁵ GitHub - en3sis/hans: An open-source, modular Discord bot template. Built with Discord.js & TypeScript

<https://github.com/en3sis/hans>

¹³ How should I structure my Discord.js v13 bot? : r/Discordjs

https://www.reddit.com/r/Discordjs/comments/1jxobmo/how_should_i_structure_my_discordjs_v13_bot/

^{17 30} Integrate APIs to Supercharge Your Discord Bot | Bootcamp Odoo

<https://bootcamp.vitraining.com/blog/python-programming-8/integrate-apis-to-supercharge-your-discord-bot-25>

³⁹ mongodb - Need advice on architecture of a Discord bot - Stack Overflow

<https://stackoverflow.com/questions/71362999/need-advice-on-architecture-of-a-discord-bot>