Especificação Técnica – Reestruturação do Log Center

Contexto e Problemas da Estrutura Atual

O **Log Center** atual apresenta diversas deficiências de arquitetura e implementação, identificadas pela equipe. Em uma análise recente, foram destacados problemas principais:

- **Práticas de logging inadequadas e falta de padronização:** Os logs não seguem um formato consistente ou uma convenção uniforme. Há ausência de bibliotecas padronizadas para geração de logs, resultando em abordagens ad-hoc em diferentes partes do projeto. Isso dificulta a manutenção e a correlação de eventos entre sistemas. A equipe enfatizou a necessidade de **padronizar os logs e a organização dos dados**, indicando que a situação atual carece de consistência.

- **Formatação de dados inconsistente:** Um exemplo grave é a prática de **armazenar conteúdo CSV como uma string dentro de um JSON** no banco de dados. Essa decisão arquitetural foi criticada por desperdiçar memória e exigir processamento extra para leitura dos dados. Conforme registrado, **uma string CSV completa estava sendo armazenada dentro de um documento JSON**, o que é *computacionalmente custoso* e pouco eficiente. O ideal seria salvar os dados já em formato JSON estruturado e somente exportá-los para CSV/Excel sob demanda, evitando conversões custosas durante a operação normal.

- **Decisões arquiteturais problemáticas:** Além do caso do CSV, notaram-se outras escolhas questionáveis. Por exemplo, determinados campos possuem nomenclatura confusa e uso indevido (foi observado que o campo `uploadedData` era utilizado para guardar um timestamp, em vez dos dados enviados). Isso revela falta de clareza no design do modelo de dados. Ademais, a aplicação foi construída sobre **Flask**, que embora funcional, não oferece suporte nativo a features modernas como *async* e escalabilidade out-of-the-box, essenciais para o crescimento do sistema. Em discussão, a equipe concluiu que FastAPI seria mais adequado, planejando **descontinuar o uso de Flask em favor do FastAPI** para novos projetos, dado seu suporte a alto desempenho e múltiplas execuções simultâneas.

- **Ausência de dashboards e relatórios adequados:** Atualmente, a forma de compartilhar informações de log com clientes ou outras partes interessadas tem sido manual e pouco amigável – por exemplo, **envio de arquivos CSV brutos** (em inglês) para clientes externos. Além da barreira de idioma, esse método é pouco prático. Foi consenso na equipe que uma interface web/dashboard onde os clientes possam visualizar informações básicas (em português, no caso) e gerar relatórios formatados (por exemplo, planilhas Excel) seria muito mais eficaz. Um dos membros reforçou que **um dashboard com informações diárias e gráficos de uso em tempo real seria o ideal** para atender às demandas dos clientes, evidenciando a defasagem do modelo atual.

Em resumo, o Log Center atual sofre com falta de padronização nos logs, formatação ineficiente de dados, arquitetura monolítica desatualizada e carência de ferramentas de visualização/administração dos dados de log. Esses pontos motivam a reestruturação completa do sistema, alinhando-o com práticas modernas de desenvolvimento.

Referência a Práticas Modernas (ComfyUI, Skyn, Kapo)

Projetos recentes dentro e fora da organização servem de referência para a modernização do Log Center. Em particular, citam-se os projetos **ComfyUI**, **Skyn** e **Kapo**, que adotaram abordagens mais modernas:

- **Frameworks Web Modernos (FastAPI):** Diferentemente do Log Center legado, que usa Flask, esses projetos optaram pelo **FastAPI** como base do backend. FastAPI oferece suporte assíncrono, melhor performance e uma estrutura de código mais modular. A decisão da equipe de padronizar novos projetos em FastAPI reflete essa tendência. Com FastAPI, obtém-se documentação automática (OpenAPI), validação de dados via Pydantic e facilidade para escalonar serviços, atendendo a múltiplas requisições simultâneas.

- **Estruturação Modular e Manutenibilidade:** Projetos modernos são estruturados em módulos ou serviços separados, seguindo princípios de *clean architecture*. Por exemplo, observa-se separação clara entre camadas de roteamento, lógica de negócio e acesso a dados. No **ComfyUI**, há uma distinção entre a interface de usuário e um backend robusto, permitindo evoluir componentes independentemente. No caso do **Skyn**, houve o aprendizado de evitar reutilização cega de código legado desatualizado, preferindo refatorar com bibliotecas atualizadas para prevenir inconsistências. Esses projetos demonstram que um código modular, com componentes bem definidos e atualizados, reduz a incidência de bugs e facilita a evolução.

- **Logging Padronizado e Observabilidade:** Todos esses projetos adotam logging consistente. Em vez de prints dispersos ou formatos arbitrários, utiliza-se uma configuração central de logging (por exemplo, integrando com a biblioteca `logging` do Python ou similares em outras linguagens) com formatação unificada – muitas vezes em JSON ou texto estruturado – incluindo timestamp, nível de severidade, origem e mensagem. Essa padronização simplifica depuração e análise de eventos em produção. Além disso, identificadores de correlação (como UUIDs ou IDs de requisição) são usados para ligar eventos relacionados através de sistemas, algo planejado para o novo Log Center também.

- **Dashboards Administrativos e Relatórios em Tempo Real:** O projeto **Kapo** destacou-se por fornecer um painel administrativo rico em informações, com atualizações em tempo real. Em vez de depender de arquivos estáticos (CSV), Kapo oferece gráficos dinâmicos, contadores e relatórios que refletem os dados conforme são gerados. Isso aumenta a transparência e a utilidade dos dados de log para a equipe e clientes. A intenção para o Log Center reestruturado é similar: construir um **dashboard web** que permita visualizar métricas de uso diário, contagem de eventos, gráficos temporais e outros indicadores, bem como exportar relatórios resumidos sob demanda. A experiência do Kapo sugere também o uso de tecnologias de *push* (e.g. WebSockets ou SSE) para notificar o frontend em

tempo real conforme novos logs chegam, garantindo que o painel seja sempre atualizado instantaneamente.

Em síntese, esses projetos fornecem um *benchmark* de qualidade: utilização de **FastAPI com arquitetura escalável**, **código modular**, **logging estruturado** e **interfaces de análise de dados em tempo real**. Com base neles, definimos a seguir as diretrizes técnicas para a reestruturação do Log Center.

Diretrizes Técnicas Propostas para o Novo Log Center

Com base na análise acima, esta seção apresenta a especificação técnica formal para reestruturar o Log Center. O objetivo é criar um **serviço de logging multiplataforma** robusto, acompanhado de um painel administrativo moderno e uma base sólida para evoluções futuras. A especificação cobre arquitetura do sistema, detalhes de implementação e boas práticas de desenvolvimento.

1. Visão Geral da Nova Arquitetura

O novo Log Center será composto de dois componentes principais: **(a) um serviço central de logging (backend)** acessível via API e **(b) um painel de administração (frontend)** para visualização e gerenciamento dos logs. Além disso, serão fornecidas **bibliotecas cliente** dedicadas para diversas plataformas (Python, JavaScript e Unity/C#), facilitando a integração de aplicações diversas ao serviço de logging de forma padronizada.

- **Serviço de Logging (Backend):** Implementado em Python usando FastAPI, expõe endpoints RESTful (e possivelmente WebSocket endpoints para streaming de logs) para receber e consultar logs. Este serviço será stateless (sem estado na aplicação, além do banco de dados), facilitando escalonamento horizontal via containers Docker. Ele incluirá lógica de autenticação/autorização básica se necessário (por exemplo, uso de tokens ou chaves de API por projeto cliente) para controlar o envio e acesso aos logs.

- **Painel de Administração (Frontend):** Aplicação web (single-page application ou server-rendered) que consome a API do Log Center. Permitirá aos usuários (internos e/ou clientes autorizados) visualizar dashboards com métricas em tempo real, filtrar e buscar logs, além de gerar relatórios (download de Excel/CSV). O painel pode ser desenvolvido em uma framework web moderna (por exemplo, React, Vue, Angular) ou como templates HTML via Jinja2/FastAPI, dependendo dos recursos da equipe. O importante é que ele forneça **visualização em tempo real** – por exemplo, exibindo novos eventos instantaneamente à medida que são registrados, possivelmente usando **WebSockets** para atualização push.

- **Bibliotecas Cliente (Python, JS, C#):** Cada biblioteca agirá como um *logger* que os projetos poderão usar para enviar eventos ao serviço central. Essas libs encapsularão os detalhes de comunicação (chamada HTTP à API, formatação JSON dos dados, etc.) e implementarão as convenções de log definidas (incluindo geração de UUID, anexação de tags padrão, etc.). Assim, um desenvolvedor poderá, por exemplo, fazer `logger = LogCenterClient(api_key="XYZ"); logger.log_event(...);` em qualquer plataforma suportada, sem se preocupar

com o protocolo. Isso garante que todos os sistemas (Python backend, aplicações web JS, aplicações Unity/C#) reportem logs de forma **consistente e padronizada**.

A comunicação entre clientes e serviço se dará predominantemente via HTTP(S). Em cenários de alto volume ou necessidades específicas, poderemos avaliar protocolos adicionais (e.g. UDP syslog, gRPC), mas inicialmente o enfoque é em **APIs RESTful bem definidas** sobre HTTP por simplicidade e ampla compatibilidade.

A seguir, detalhamos cada aspecto técnico da proposta.

2. Estrutura de Repositório (FastAPI + Docker)

A reestruturação adotará uma organização de repositório inspirada nas melhores práticas de projetos FastAPI e containerização. Julio ficará responsável por preparar o repositório do Log Center com essa estrutura modernizada. A estrutura proposta é:

```
None
logcenter/
├── app/
│   ├── main.py          # Inicialização da aplicação FastAPI, instancia app e
inclui rotas
│   ├── api/             # Pacote com módulos de rotas (routers)
│   │   ├── logs.py      # Roteador para endpoints de logs (upload, query, etc.)
│   │   ├── projects.py  # (Opcional) roteador para gerenciar projetos/configurações
│   │   └── ...
│   ├── models/          # Definições de modelos de dados (Pydantic models, schemas)
│   │   ├── log.py       # Schema Pydantic para LogEntry, etc.
│   │   └── ...
│   ├── services/        # Lógica de negócio (por ex., funções de processamento,
agregação)
│   ├── db/              # Configuração do banco de dados (conexão, ORMs/ODM if any)
│   ├── core/            # Configurações centrais (auth, middleware, config)
│   └── util/            # Utilitários (ex: gerador de UUID, funções de tempo, etc.)
├── tests/               # Testes unitários e de integração
├── docker-compose.yml   # Definição dos serviços (API, banco de dados, etc.) para
desenvolvimento
├── Dockerfile           # Dockerfile para build da imagem do serviço FastAPI
├── .env.example         # Exemplo de configuração de ambiente (chaves, URIs...)
└── README.md            # Documentação básica do projeto
```

Detalhes dessa organização:

- Os **routers do FastAPI** ficam segregados por domínio (por exemplo, `logs.py` para operações relacionadas a logs: envio, consulta, relatório; poderíamos ter outros como `auth.py` se houver autenticação, etc.). Cada router será incluído no `main.py` dentro de instâncias *APIRouter* correspondentes, semelhante à forma como Blueprints eram usados no Flask.

- Os **modelos Pydantic** em `models/` definem os esquemas JSON de entrada e saída. Por exemplo, um modelo `LogEntryCreate` para validar os dados ao receber um novo log, e um modelo `LogEntry` para retornar dados ao cliente. Isso garante que

os dados trafegando pela API estejam no formato esperado e documenta claramente o contrato da API.

- A pasta **services/** conterá lógica de aplicação independente de framework (por exemplo, funções para agregar logs, gerar estatísticas, aplicar filtros). Isso segue o princípio de separar a lógica de negócio dos detalhes de transporte (os routers), facilitando testes e reuso.

- Em **db/** teremos a configuração da base de dados. Planeja-se usar a abordagem de *driver nativo* do MongoDB (pymongo ou motor async do Mongo). Alternativamente, podemos considerar um ODM (Object Document Mapper) ou até ORMs multi-banco (como SQLAlchemy) para maior flexibilidade, mas dado o escopo de logs e a escolha primária por MongoDB, inicialmente uma configuração simples do cliente do Mongo deve bastar. Ainda assim, abstrairemos o acesso ao banco de forma que seja possível trocar ou adicionar um banco relacional no futuro sem grandes impactos (mais detalhes na seção de Banco de Dados).

- O **Dockerfile** será configurado para criar a imagem do servidor FastAPI, seguindo uma base Python 3.11 (ou versão atual estável) slim. Incluirá as dependências listadas (possivelmente gerenciadas via Poetry ou requirements.txt) e usará o Uvicorn como servidor ASGI. O **docker-compose.yml** facilitará o desenvolvimento local, orquestrando contêineres do app e do MongoDB. Em produção, pode-se usar imagens separadas (um deployment Kubernetes ou serviço de containers), mas ter o Dockerfile permite portabilidade. A adoção do Docker padroniza o ambiente de execução, evitando problemas de configuração em diferentes máquinas.

- Serão mantidos exemplos de configuração no *.env.example* (como string de conexão do Mongo, credenciais default, etc.) e o README.md documentará como subir o ambiente, executar testes e publicar logs via API.

Essa estrutura modular facilitará a colaboração (cada desenvolvedor pode atuar em um módulo isoladamente), a **escalabilidade do desenvolvimento** (adicionar novas funcionalidades sem quebrar outras) e a aderência às decisões arquiteturais combinadas (FastAPI, Docker, etc.). Também torna mais simples a **adoção de padrões** – por exemplo, inclusão de middlewares de logging ou CORS no startup, conforme necessário.

3. Esquema de Dados Padronizado (JSON, UUIDs e Tags)

Um dos pilares da reestruturação é a definição de um **esquema de dados consistente** para os logs. Todas as informações de log deverão ser armazenadas e transmitidas em **formato JSON**, aproveitando a flexibilidade e legibilidade desse formato. A especificação do esquema é a seguinte:

Modelo de Log (LogEntry):

Cada entrada de log será representada por um objeto JSON com campos padronizados. Abaixo um exemplo de JSON de um log e descrição de seus campos:

```
JSON
{
    "id": "a45f1c2e-3b67-4f9d-9e8a-2d48ec6f1b23",           // UUID v4 do evento de
log
    "timestamp": "2025-09-01T17:46:20Z",                   // Data/hora do evento em
formato ISO 8601 (UTC)
    "project": "RedBullMachine",                           // Identificador do
projeto/sistema de origem
    "level": "INFO",                                       // Nível de severidade
(INFO, WARNING, ERROR, etc.)
    "tags": ["dispensador", "operacao"],                   // Lista de tags
categorizando o evento
    "message": "Ciclo de dispensa completado",          // Descrição breve do evento
    "data": {                                              // Objeto JSON com dados
adicionais (payload) do evento
        "sessionId": "XYZ123",
        "itemsDispensed": 5,
        "operator": "João"
    }

    "request"
}
```

- **`id` (UUID):** Identificador único universal para cada log. Será gerado no momento em que o log é criado (pelo cliente ou pelo servidor) utilizando UUID v4. **Todas as bibliotecas cliente gerarão um UUID** para identificar o evento antes do envio, garantindo unicidade mesmo que múltiplas fontes reportem simultaneamente. Esse UUID facilita a busca e indexação de logs específicos, além de possibilitar correlação de eventos entre sistemas (por exemplo, um mesmo id log podendo ser referenciado em subsistemas diferentes, caso se propague).

- **`timestamp`:** Momento em que o evento ocorreu, no padrão ISO 8601 UTC (Zulu). O timestamp preferencialmente será fornecido pelo cliente (para refletir o momento exato no dispositivo/origem). Contudo, o servidor também pode sobrescrever ou validar para garantir ordenação consistente (por exemplo, registrando um `received_at` separado se necessário). A formatação padronizada simplifica ordenação e cálculo de intervalos de tempo.

- **`project`:** Nome ou identificador do projeto/aplicação de onde o log se originou. Como o serviço será multi-plataforma e centralizado, é crucial rotular a origem. Este campo pode ser um nome amigável ("RedBullMachine") ou um código/UUID do projeto definido previamente no sistema. O Log Center pode manter uma

coleção de **projetos cadastrados** para validar este campo e associar metadados (e.g., chave de API do projeto para autenticação, descrição, etc.).

- **`level`:** Nível de severidade ou tipo do log. Valores comuns: `"DEBUG"`, `"INFO"`, `"WARNING"`, `"ERROR"`, `"CRITICAL"`. Podemos adotar a convenção da biblioteca Python `logging` (que também é usada em muitas outras linguagens) para manter familiaridade. Embora não fosse utilizado explicitamente na versão antiga (que tinha um campo `status` pouco esclarecedor), adotar níveis padronizados permitirá filtragem fácil de eventos no dashboard (ex.: ver apenas erros). Bibliotecas cliente deverão mapear adequadamente seus níveis de log para esses valores.

- **`tags`:** Lista de etiquetas categorizando o evento. Diferente de `level`, que indica severidade, **tags** podem descrever a natureza do evento, subsistema ou qualquer categoria relevante. Por exemplo, tags poderiam indicar módulos (`"network"`, `"UI"`, `"sensor"`), fases (`"startup"`, `"shutdown"`), ou características do log (`"performance"`, `"security"`). Haverá uma **lista de padrões de tags recomendadas** no guideline de desenvolvimento, para evitar proliferação descontrolada; porém, tags customizadas serão permitidas para abranger casos específicos. Tags facilitam buscas avançadas – por exemplo, filtrar todos os logs de `"operacao"` e `"erro"` no painel para ver falhas operacionais.

- **`message`:** Uma descrição textual resumida do evento. Deve ser suficiente para entender o que ocorreu (e.g., "Usuario X realizou login", "Temperatura acima do limite"). Idealmente em português se for voltado a operadores internos, ou em inglês técnico se for apenas para desenvolvedores – a decisão aqui deve acompanhar quem é o público consumidor do log. No contexto do dashboard para clientes, possivelmente o `message` poderia ser traduzido ou padronizado em PT-BR para facilitar compreensão. Essa frase não deve ser extremamente longa; detalhes adicionais vão em `data`.

- **`data`:** Objeto JSON opcional contendo **dados adicionais** do evento. Aqui pode-se incluir informações estruturadas relevantes que complementem o `message`. Por exemplo, no log de um dispenser de produtos, `data` pode conter quantos itens foram dispensados, qual o ID da sessão, temperatura de operação, etc. Ao contrário da implementação antiga, onde um log complexo podia virar uma linha CSV inteira armazenada como texto bruto, agora esses detalhes serão guardados em campos devidamente tipados dentro de `data`. Isso permite consultas e agregações diretas no banco (e.g., somar `itemsDispensed` ao longo do dia) e simplifica a conversão para outros formatos (uma simples exportação para CSV ou Excel das chaves do JSON, se requisitado).

Validação e Conversão: Com o uso de Pydantic, a API validará automaticamente esses campos ao receber um log. Campos obrigatórios (por exemplo, `id`, `timestamp`, `project`, `message`) serão exigidos; outros podem ser opcionais (por exemplo, `data` pode ser omitido se não houver informações extras). A conversão de tipos (ex.: string de timestamp para objeto datetime) será feita no servidor para armazenamento adequado. No banco de dados MongoDB, os documentos seguirão esse esquema; poderemos aproveitar o ObjectId do Mongo como chave primária, mas o campo `id` (UUID) será armazenado separadamente para manter o vínculo externo.

Exportação e Formatos Derivados: Caso seja necessário exportar logs para outros formatos (CSV, Excel), o servidor implementará rotinas de conversão a partir desses documentos JSON. Por exemplo, uma solicitação de relatório pode produzir um CSV com colunas correspondentes aos campos acima (cada chave JSON vira uma coluna). Isso atende à necessidade de fornecer relatórios em formatos que os clientes entendam (como Excel), sem sacrificar a eficiência do armazenamento interno. O importante é que o **armazenamento principal permaneça em JSON estruturado**, garantindo eficiência e flexibilidade.

4. Banco de Dados (MongoDB como Primário, Extensível a SQL)

Optou-se por utilizar o **MongoDB** como banco de dados primário do Log Center reestruturado. Essa escolha se deve à natureza dos dados de log – sem esquema fixo rigoroso, volume potencialmente alto, necessidade de inserções rápidas e consultas flexíveis – áreas onde MongoDB (um banco de dados orientado a documentos) se destaca.

Estrutura no MongoDB: Iremos modelar uma ou mais coleções conforme a necessidade:

- Coleção principal de **logs** (por exemplo, chamada `logs` ou `log_entries`): armazenará os documentos JSON de cada evento, seguindo o esquema padronizado descrito. Os campos relevantes (como `project`, `timestamp`, `tags`, `level`) serão indexados para permitir buscas eficientes. Por exemplo, é recomendável um índice composto por `project + timestamp` para extrair logs de um determinado projeto ordenados por data; índices em `tags` (provavelmente um índice multi-chaves, já que tags é um array) para permitir filtros por categoria; e índice em `id` (UUID) caso se queira buscar diretamente por um log específico.

- Coleção de **projetos** (ex: `projects`): guardará informações sobre cada sistema/cliente que envia logs. Campos podem incluir um `project_id` (talvez um UUID para identificar projetos), nome, descrição, e configurações como chave de API, preferências de retenção, etc. Isso permite validar o campo `project` nos logs e agregar dados por projeto facilmente. Também pode armazenar, por exemplo, quais tags ou níveis cada projeto utiliza, se for útil para o painel filtrar configurações.

- Coleção de **usuários/admin** (se for necessário controlar acesso ao painel): para autenticação no painel de administração, poderíamos ter usuários com permissões, associados a certos projetos (de modo que um cliente só veja logs do seu projeto). Entretanto, detalhes de autenticação não foram foco dessa especificação, podendo inicialmente assumir uso interno somente.

Extensibilidade para SQL/Relacional: Embora MongoDB seja adequado para o cenário inicial, considera-se a possibilidade de alguns dados ou funcionalidades requererem um banco relacional no futuro (por exemplo, integração com sistemas legados, ou necessidade de *joins* complexos com outras bases). Para preparar terreno para isso, adotaremos as seguintes estratégias:

- **Abstração de Camada de Dados:** Implementaremos as operações de banco de forma abstrata (por exemplo, via uma interface/repositório). Em vez de chamar

diretamente métodos do `pymongo` espalhados pelo código, teremos funções centralizadas como `save_log(log_entry)` ou `query_logs(filter)` dentro do módulo de dados. Assim, se no futuro precisarmos mudar a implementação para usar um Postgres ou outro banco, essa troca fica isolada nesse módulo.

- **Uso Condicional de ORMs/ODMs:** Podemos utilizar uma biblioteca como **SQLAlchemy** em paralelo, caso seja necessário gravar alguns dados em um SQL. Por exemplo, armazenar um resumo diário de logs (agregações) em uma tabela relacional para facilitar integração com ferramentas BI externas. O projeto será estruturado para não limitar tal adição – ou seja, não utilizaremos recursos específicos do Mongo que não tenham equivalente no SQL sem pelo menos documentar uma alternativa. Por exemplo, para busca textual em logs, podemos usar tanto o *full text search* do Mongo quanto manter um índice em ElasticSearch ou Postgres (caso haja demanda futura por consultas mais elaboradas). A prioridade, contudo, é fazer o MongoDB atender plenamente as necessidades de logs brutos, usando relacional apenas se surgirem requisitos que o justifiquem.
- **Migração de Dados:** Se for necessário migrar parte ou todos os dados de log do Mongo para um relacional, providenciaremos scripts de exportação. Dado que os logs estão em JSON, é possível mapeá-los para colunas tabulares (cada tag vira uma coluna booleana? ou usar JSON nativo do Postgres, etc.). Essa migração não faz parte do escopo inicial, mas a modelagem não irá impedir isso futuramente.

Em termos operacionais, usando Docker, podemos incluir um serviço do MongoDB (por exemplo, imagem oficial do MongoDB) no docker-compose para desenvolvimento. Em produção, o MongoDB pode ser gerenciado via um cluster existente ou serviço gerenciado, conforme o caso. Importante será configurar parâmetros de produção: TTL indexes ou políticas de expiração (se desejarmos purgar logs antigos após X dias para controle de crescimento), replicação para alta disponibilidade e backups regulares. Tais detalhes operacionais ficarão na documentação de implantação.

5. Serviço de Logging Multiplataforma (Backend FastAPI)

O serviço principal de logging será uma aplicação FastAPI, responsável por expor endpoints para recebimento de logs, consulta e gerenciamento básico. Aqui delineamos os principais endpoints e comportamentos:

Endpoints de Recebimento de Logs:

- `POST /api/v1/logs` – Endpoint para envio de um novo log. Os clientes (via bibliotecas ou diretamente) realizarão requisições POST contendo um JSON conforme o esquema definido. Exemplo de corpo da requisição:

```
JSON
{
  "id": "a45f1c2e-3b67-4f9d-9e8a-2d48ec6f1b23",
  "timestamp": "2025-09-01T17:46:20Z",
```

```
  "project": "RedBullMachine",
  "level": "INFO",
  "tags": ["dispensador", "operacao"],
  "message": "Ciclo de dispensa completado",
  "data": { "...": "..." }
}
```

- O servidor irá validar os campos (respondendo com erro 422 em caso de formato inválido ou campos faltantes). Uma vez validado, inserirá o documento no MongoDB. Em caso de sucesso, retornará código 201 Created, possivelmente com um body contendo o `id` ou URL do recurso criado.

Detalhe: Podemos considerar permitir envio em **lote** (batch) – por exemplo, `POST /logs` aceitando uma lista de logs – para eficiência quando um cliente precisar enviar muitos eventos acumulados (isso pode acontecer se o dispositivo ficou offline e armazenou logs localmente). Na V1, podemos focar no envio unitário e adicionar batch posteriormente.

- **Autenticação/Autorização:** Inicialmente, se o Log Center for usado internamente, poderíamos não exigir autenticação no endpoint de log. Porém, para uso por clientes externos ou múltiplos projetos, é recomendável implementar uma forma de autenticar essas requisições para evitar spam ou mistura de dados. Uma abordagem leve: fornecer a cada projeto uma **API Key** (um token) e exigir um header, por exemplo `X-API-Key: <token>`. O servidor validaria correspondendo o token ao projeto informado no JSON (cada projeto teria seu token armazenado). Assim, evitamos que um projeto envie dados se não apresentar credencial válida. Esse esquema pode evoluir depois para OAuth ou outro mecanismo se necessário, mas manteremos simples no início.

Endpoints de Consulta e Relatório:

- `GET /api/v1/logs` – Retorna logs armazenados, possivelmente com parâmetros de query para filtros:

- Filtros possíveis: `?project=NomeProj` (filtrar por projeto), `?level=ERROR` (filtrar por nível), `?tag=operacao` (filtrar logs que contenham certa tag), `?start=2025-09-01T00:00:00Z&end=2025-09-01T23:59:59Z` (filtrar por intervalo de tempo), etc.

- Suporte a paginação: e.g. `?page=2&page_size=100` ou um esquema de paginação por cursor (p.ex., `?after_id=<uuid>`).

- O resultado seria uma lista de objetos JSON de log (conforme esquema), provavelmente limitada em quantidade (para não retornar milhões de logs de uma vez). Exemplo de resposta:

```JSON
{
  "results": [ {<log1>}, {<log2>}, ... ],
  "count": 1520,
  "page": 2,
  "page_size": 100
}
```

- O servidor fará as consultas no Mongo aplicando os filtros (provavelmente convertendo parâmetros em condições no find, usando índices adequados). Antes de retornar, converterá alguns campos se necessário (e.g., datas para string ISO). *Nota:* no protótipo atual do Log Center (Flask), havia um endpoint `/datalog` GET que retornava todos os logs e fazia conversão de ObjectId para string e datetimes para string ISO. Usaremos essa mesma ideia, mas com possibilidade de filtros e paginação, tornando a busca mais útil.

- `GET /api/v1/logs/<id>` – Consulta um log específico pelo UUID (ou pelo ObjectId do Mongo). Retorna o objeto do log ou 404 se não encontrado. Útil para buscar detalhes de um evento específico, talvez referenciado em algum alerta.

- **Endpoints agregados / dashboards:** Para alimentar os dashboards em tempo real e relatórios, o backend também pode fornecer endpoints que retornem **estatísticas agregadas**. Por exemplo:

- `GET /api/v1/stats/daily?project=ProjX&date=2025-09-01` – retorna contagem de eventos por nível naquele dia, total de ocorrências de certos tags, etc.

- `GET /api/v1/stats/timeseries?project=ProjX&interval=hour` – retorna uma série temporal (e.g., logs por hora nas últimas 24h).

Esses endpoints condensam informações e evitam que o frontend tenha que baixar todos os logs para computar localmente. Com o uso de *aggregation pipeline* do MongoDB, podemos computar muitos desses dados eficientemente no servidor. Por exemplo, para um gráfico de utilização por horário, um pipeline Mongo agrupando por hora de timestamp e contando documentos já resolve.

- **Streaming de Logs:** Uma funcionalidade a considerar é permitir ao painel receber novos logs em tempo real sem polling constante. O FastAPI suporta **WebSockets**, então poderíamos ter `GET /api/v1/logs/stream` via WebSocket, onde o cliente (dashboard) se conecta e o servidor envia cada novo log inserido (filtrado por projeto, possivelmente) ao cliente imediatamente. Internamente, isso pode ser implementado inscrevendo o websocket em notificações de novo log (via um sistema pub/sub simples ou mesmo postando para websockets em cada inserção). Esta é uma funcionalidade avançada – se não implementada inicialmente, o dashboard pode começar atualizando via consultas periódicas (e.g., a cada 5 segundos) até evoluirmos para push.

Todos os endpoints serão documentados conforme o padrão OpenAPI gerado pelo FastAPI, permitindo fácil inspeção e testes (uma UI Swagger redirecionada em `/docs`). Isso reforça a **usabilidade do serviço** tanto para desenvolvedores integrando quanto para manutenção futura.

6. Bibliotecas Cliente (Python, JavaScript, C# Unity)

Para assegurar que o log seja usado de forma consistente em múltiplas plataformas, serão disponibilizadas bibliotecas cliente oficiais do Log Center para as linguagens alvo:

- **Python Client Library:** Distribuída via PyPI (nome sugestivo: `logcenter-sdk`), compatível com Python 3. A biblioteca fornecerá uma classe principal (por exemplo, `LogCenterClient`) que encapsula as chamadas HTTP ao serviço. Permitirá configurar a URL do servidor, chave de API do projeto e outros parâmetros. Também pode oferecer métodos correspondentes aos níveis de log (ex: `log_info(msg, data=None, tags=None)`, `log_error(msg, ...)`), que internamente montarão o JSON e enviarão via `requests` (ou `httpx`/asyncio se for ser usada assincronamente). Essa lib poderá integrar-se com a biblioteca standard `logging` do Python através de um `Handler` customizado – por exemplo, o desenvolvedor pode adicionar um `LogCenterHandler` ao seu logger, de modo que qualquer chamada `logging.info("...")` automaticamente envie para o Log Center. Esse tipo de integração torna a adoção transparente nas aplicações Python.
    - Touchdesigner TODO
- **JavaScript Client Library:** Publicada talvez via npm (nome: `logcenter-js`). Suportará aplicações **web** e **Node.js**. No contexto de aplicações web, seria usada para enviar logs de front-end (como interações do usuário, erros de UI, métricas de performance no navegador) para o Log Center. Deverá funcionar de forma assíncrona sem travar a interface. No caso de Node.js (ou serviços backend em JS), igualmente facilitaria reportar eventos do servidor. A biblioteca usará `fetch` ou `axios` conforme o contexto para fazer as requisições. Em aplicações web, deve-se cuidar para não expor indevidamente a API Key – possivelmente, será necessário um modelo no qual o front-end se autentica de outra forma (ou só envie logs não sensíveis publicamente). De início, podemos focar no uso interno ou em ambientes controlados.
- **Unity C# Client Library:** Desenvolvida em C# (possivelmente como um pacote Unity ou uma DLL padrão) para integrar em aplicações Unity (por exemplo, kiosks, jogos ou simulações que a empresa desenvolve). Essa biblioteca usará as classes de rede do .NET/Unity (como `UnityWebRequest`) para enviar os logs. Como Unity pode rodar em ambientes variáveis (desktop, mobile, etc.), a biblioteca deve ser robusta a falhas de conexão – talvez enfileirando logs quando offline e enviando quando reconectar. Dado que Unity muitas vezes lida com logs de jogos em tempo real, essa biblioteca deve ter baixo overhead para não impactar o frame rate; idealmente, envio em segundo plano.

Funcionalidades comuns das bibliotecas: Todas as bibliotecas seguirão as mesmas convenções:

- **Configuração Inicial:** O desenvolvedor deverá inicializar a lib com pelo menos o endpoint do servidor e a identificação/autenticação do projeto (por ex., API Key). Poderá haver configurações adicionais como: modo offline (buffering), nível mínimo de log a enviar (e.g., ignorar DEBUG em produção), etc.
- **Geração de UUID e Timestamp:** As libs irão gerar o UUID do log event e capturar o timestamp local automaticamente, caso o desenvolvedor não forneça. Isso alivia o trabalho de quem usa a biblioteca e garante que nenhum log fique sem essas informações básicas. (O servidor ainda pode validar e ajustar, mas em geral usará os fornecidos).

- **Assinatura de *interface* semelhante:** Exemplo pseudo-código em:
    - Python:

```python
client = LogCenterClient(api_key="MINHA_KEY", project="RedBullMachine")
client.log("INFO", "Ciclo completado", data={"items":5},
tags=["operacao","dispensador"])
```

    - Em JavaScript:

```javascript
const client = new LogCenterClient({ apiKey: "MINHA_KEY", project:
"RedBullMachine" });
client.log("INFO", "Ciclo completado", { items: 5 },
["operacao","dispensador"]);
```

    - Em C#:

```csharp
var client = new LogCenterClient("https://logcenter.api", "MINHA_KEY",
"RedBullMachine");
client.Log("INFO", "Ciclo completado", new { items = 5 }, new string[]{
"operacao", "dispensador" });
```

- Em todos os casos, a biblioteca formatará estes parâmetros num JSON exatamente como especificado pelo esquema, e fará o HTTP POST ao endpoint.
- **Erros e Retentativas:** As libs deverão tratar erros de forma apropriada. Se o envio falhar (por ex., servidor indisponível ou sem conexão), podem implementar retentativas exponenciais ou armazenar em um buffer local (arquivo, memória) e tentar novamente depois de alguns segundos. Isso é especialmente importante para dispositivos Unity ou front-ends que podem temporariamente ficar offline. Assim evitamos perda de logs. Além disso, as bibliotecas podem optar por executar de forma assíncrona as requisições (no caso de Python, poderia oferecer métodos async ou rodar em threadpool, em JS usar promises/async, em C# usar async/await) para não bloquear a aplicação principal.
- **Segurança:** As bibliotecas devem usar HTTPS para comunicar com o servidor em produção, garantindo que dados (que podem incluir informações sensíveis de operação) não trafeguem em texto plano. Além disso, como mencionado, proteger as chaves de API – em apps Unity, por exemplo, pode-se ofuscar a chave ou usar um mecanismo de obtenção segura; em apps web, considerar limites de uso e monitoramento, já que a chave poderia ficar exposta no JavaScript (nesse caso,

possivelmente logs públicos não críticos podem ser permitidos sem autenticação explícita, ou usar um token de sessão atrelado a um usuário logado no sistema).

Em suma, as bibliotecas cliente são parte integral do ecossistema Log Center reestruturado, provendo **facilidade de integração** e **garantindo padronização** de formato de log em todas as linguagens alvo. Isso atende diretamente à meta de criar *"um sistema de log mais consistente e fácil de usar"* em toda a empresa.

7. Painel de Administração e Dashboards em Tempo Real

O **painel de administração** do Log Center será a face visível dos dados de log, voltado tanto para a equipe interna acompanhar o comportamento dos sistemas quanto, possivelmente, para clientes finais monitorarem indicadores de suas aplicações. Inspirado pelo projeto Kapo, esse painel oferecerá **dashboards personalizáveis e relatórios em tempo real**, de forma intuitiva.

Tecnologia e Implementação: Podemos implementar o painel de duas maneiras – **integrado** ou **separado**:

- **Integrado ao FastAPI (monolito):** Aproveitar o próprio FastAPI para servir páginas HTML (usando Jinja2 templates) ou servir um aplicativo front-end estático. Nesse cenário, o repositório do Log Center incluiria uma pasta `frontend` com uma aplicação (por ex. em React) que é construída e cujos arquivos estáticos (HTML, JS, CSS) são servidos pelo FastAPI (via `StaticFiles`). Isso facilita o deployment (um único serviço) e garante coerência de versões entre API e frontend.

- **Aplicação Separada:** Manter o front-end em um projeto separado, especialmente se usar um framework front-end robusto. Este front-end consumiria a API do Log Center remotamente. Isso adiciona complexidade de implantação (dois serviços), mas isola melhor as responsabilidades (um time pode evoluir o dashboard sem tocar no backend, e vice-versa). Dado que a equipe mencionou *"painel semelhante ao Kapo"*, podemos inferir que valeria investir em um front-end rico. Para esta especificação, detalharemos as funcionalidades esperadas, independente da escolha de implementação.

Funcionalidades do Dashboard:

- **Autenticação e Controle de Acesso:** O painel terá login de usuário (ou ao mínimo, proteção por senha) para evitar acesso público aos dados de log. Poderá integrar com o sistema de autenticação da empresa ou ter usuários próprios. Também podemos implementar controle de acesso por projeto – por exemplo, um usuário cliente só enxerga logs e gráficos do projeto dele, enquanto usuários administradores veem de todos.

- **Visão Geral (Overview):** Uma página inicial mostrando indicadores-chave:

- Número de eventos nas últimas 24h (por projeto ou total).

- Porcentagem de eventos por nível (ex: 5% errors, 15% warnings, 80% info).

- Gráfico de linha ou barras mostrando quantidade de logs por intervalo de tempo (últimas horas/dias).
- Indicadores personalizáveis conforme o tipo de projeto – por exemplo, para um sistema de vending machine, mostrar *“Produtos dispensados hoje”*, *“Média por hora”*, etc., usando dados dos campos específicos dos logs (`data.itemsDispensed` agregado, etc.).
- **Visão de Logs em Tempo Real:** Uma seção tipo *log tail*, onde os eventos recentes aparecem em uma tabela atualizada em tempo real. Similar a ver um arquivo de log crescendo, mas filtrado e formatado:
- Colunas como Timestamp, Projeto, Nível, Mensagem, e talvez ações (ver detalhes).
- Capacidade de filtrar na hora por nível (botões para mostrar/ocultar INFO, DEBUG, etc.), por texto (buscar palavra-chave na mensagem), por tags (checkbox de tags comuns).
- Atualização em tempo real: se um novo log chega que corresponde ao filtro atual, ele é inserido no topo da lista automaticamente (usando WebSocket ou polling curto).
- A tabela pode ter paginação ou carregamento infinito, mas o foco é nos recentes. Um botão *"Pausar"* pode congelar a atualização se o usuário quiser inspecionar algo sem interrupção.
- **Detalhe de Log:** Ao clicar num log específico na tabela, abrir um modal ou página de detalhes, mostrando todos os campos, inclusive o objeto `data` formatado legivelmente (por exemplo, em JSON indentado ou em tabela de chave-valor). Isso permite investigar um evento profundamente. Também poderia oferecer ações como *“Exportar este log (JSON)”* ou *“Copiar ID”*.
- **Dashboards Específicos:** Além da visão geral, poderemos ter abas de dashboards focados:
- **Por Projeto:** Selecione um projeto e veja métricas específicas dele – e.g., uptime (se houver logs de heartbeat), uso por hora, erros frequentes, etc. Possibilidade de comparar projetos (se for relevante).
- **Por Tag ou Módulo:** Por exemplo, um dashboard só de logs de *"performance"*, mostrando tempos médios, picos, etc., se coletados.
- **Alertas e Tendências:** Configurar algumas condições (ex: >X erros por hora) e o painel destaca ou envia notificação (no próprio UI ou email). Isso é um extra que poderia ser adicionado para tornar o Log Center não só passivo, mas ativo na monitoração.
- **Relatórios:** Concretizando a recomendação da equipe, o painel permitirá exportar dados em formatos *amigáveis ao usuário*. Em vez de enviar CSV cru em inglês, o usuário poderá:

- Escolher um período e gerar um relatório (por exemplo, "Logs de Agosto 2025") e baixar um arquivo Excel (.xlsx) ou PDF com formatação. Esse relatório pode incluir gráficos e tabelas resumitivas, além dos dados brutos se necessário.

- Alternativamente, exportar apenas os dados filtrados visíveis na tela para CSV/Excel. Assim, se o cliente filtrou "julho, erros", ele obtém só aquilo.

- Os relatórios gerados podem ser bilíngues ou configurados em português para o cliente final, seguindo a sugestão de fornecer conteúdo no idioma do usuário final.

- **Tecnologia de Gráficos e UI:** Podemos utilizar bibliotecas de gráficos JavaScript como **Chart.js**, **D3.js** ou **Recharts** para visualizar dados no frontend. Para componentes de interface (tabelas, filtros, layout), frameworks populares de UI (como Ant Design, Material UI) podem acelerar o desenvolvimento. Como desempenho é importante, principalmente para atualizações em tempo real, devemos otimizar a quantidade de dados trafegados – por exemplo, pré-agregar contagens no backend para enviar somente os pontos do gráfico em vez de todos os logs.

- **Responsividade e Acessibilidade:** O dashboard deve ser web-responsivo, permitindo visualização em diferentes tamanhos de tela (monitores grandes em NOCs, ou acesso rápido via celular/tablet se necessário). Também deve obedecer boas práticas de UX – por exemplo, mostrar claramente quando está *"Atualizado até X seg atrás"* ou se perdeu conexão com o stream de logs, etc.

O desenvolvimento do painel vai ao encontro da estratégia de melhorar a transparência dos dados para clientes e reduzir a necessidade de trocas manuais de arquivos. Conforme citado, em vez de enviar CSVs para os clientes, teremos **"uma interface onde eles possam acessar informações básicas... e baixar relatórios em Excel"**, em português e de forma auto-serviço. Adicionalmente, um *dashboard com informações diárias e gráficos por horário* foi apontado como ideal para clientes como a Eletromídia – nossa implementação visará exatamente fornecer esse nível de informação de maneira automatizada e contínua.

8. Estratégia de Retrocompatibilidade e Migração de Dados Legados

Dada a existência de dados de log históricos no sistema atual (incluindo aqueles armazenados nos formatos não ideais, como CSV embutido em JSON), é essencial planejar a migração e/ou compatibilidade retroativa para não haver perda de informação nem interrupção dos serviços durante a transição.

Migração de Dados: A primeira decisão é migrar ou não os dados antigos para o novo esquema. Considerando que: - O esquema antigo difere do novo (por exemplo, campo `uploadedData` era datetime e antes podia ter sido texto CSV), - O volume de dados pode ser considerável (avaliar quantos registros existem),

A estratégia proposta:

1. **Migração Offline:** Desenvolver um script de migração que percorra todos os documentos antigos, convertendo-os para o novo formato e inserindo-os na nova coleção de logs. Esse script pode: - Ler cada documento da coleção antiga

`datalog` (usada pelo Flask). - Para cada documento, interpretar os campos: - Se houver um campo de texto que contenha CSV (conforme mencionado no caso do sistema Red Bull), realizar o parsing CSV e transformar em estrutura JSON dentro de `data`. Por exemplo, se o CSV tinha colunas fixas, mapear para subcampos nomeados. - Converter campos como `timePlayed` e `uploadedData` para os novos `timestamp` (provavelmente `timePlayed` era o momento do evento no dispositivo e `uploadedData` a hora de upload; no novo modelo, possivelmente só precisamos de um timestamp – podemos optar por usar `timePlayed` como `timestamp` do evento e descartar `uploadedData` ou mantê-lo como um campo separado `received_at` se for relevante). - Gerar um UUID para cada registro migrado, já que os antigos não têm esse campo. Poderíamos usar o ObjectId existente para derivar um UUID de forma determinística (por exemplo, hash do ObjectId) ou simplesmente gerar um novo (v4). Como é histórico, gerar um novo não deve causar problemas, desde que apontemos que aqueles logs migrados têm uuid recém-gerado. - Aplicar tags se possível: por exemplo, se sabemos que todos logs da máquina X deveriam ter tag "operacao", talvez inferir durante a migração. No entanto, isso pode ser complexo – é provável que a maioria das tags em dados legados fique vazia ou apenas derivadas do status/level, a não ser que haja alguma indicação no conteúdo. - Mapear `status` antigo para `level` novo (se `status` era algo como "OK"/"FAIL", mapear para INFO/ERROR, etc., conforme compreensão do domínio). - Inserir o documento convertido na nova coleção. - Marcar de alguma forma que aquele registro veio do legado (talvez uma tag "legacy" ou um campo booleano). Este script seria executado uma vez, antes de colocar o novo sistema em produção, para pré-popular o novo banco.

2. **Congelamento do Sistema Antigo:** Idealmente, durante a migração, o antigo Log Center ficaria **congelado** (sem receber novos logs) ou rodando em somente-leitura. Isso para evitar divergência – caso contrário, logs poderiam entrar no sistema antigo enquanto migramos e serem perdidos. Uma estratégia é coordenar a mudança em janela de manutenção: pára-se o envio de logs (ou redireciona dispositivos para acumular localmente), executa a migração, e então inicia-se o novo sistema.

3. **Compatibilidade de API Legada:** Se houver clientes que enviam logs diretamente ao antigo endpoint (por exemplo, dispositivos configurados para chamar a API Flask atual), devemos oferecer uma transição suave. Algumas abordagens:

4. **Emular endpoints antigos no novo serviço:** Implementar no FastAPI endpoints com os mesmos paths e parâmetros que o antigo (/datalog/upload, etc.), de forma que redirecionem internamente para o novo processamento. Isso permitiria atualizar o servidor sem necessariamente atualizar todos os clientes de imediato. Esses endpoints poderiam interpretar a requisição no formato antigo (ex: multipart form com `timePlayed` e possivelmente um arquivo CSV) e então converter para o novo esquema e salvar. Assim, dispositivos legados funcionariam sem alterações. Essa camada de compatibilidade poderia ser temporária, e documentada como *deprecada*.

5. **Redirecionamento HTTP:** Alternativamente, se possível configurar, o endpoint antigo poderia retornar um redirect ou erro informando alteração, mas isso requer mudança do cliente, então talvez não ideal.

6. **Notificação e Atualização dos Clientes:** Em paralelo, é recomendável notificar desenvolvedores e responsáveis pelos sistemas clientes para **atualizarem para as novas bibliotecas** assim que possível. Como teremos libs oficiais, a migração dos clientes consistirá em trocar chamadas antigas pela nova lib, algo que deve ser facilitado pela melhoria (como as libs fazem mais trabalho automaticamente). Até lá, a compatibilidade dos endpoints garante funcionamento.

7. **Verificação Pós-Migração:** Depois de migrar os dados e passar a receber logs no novo sistema, é importante validar que nada foi perdido:

8. Conferir contagem de registros antigos vs migrados.

9. Fazer consultas pontuais nos dois sistemas para ver se batem (ex: último log de cada projeto em ambos).

10. Manter, por segurança, um backup do banco antigo (ou mantê-lo rodando em separado) por algum tempo, caso seja necessário consultar algo que não foi migrado corretamente.

11. **Desativação Gradual do Legado:** Uma vez que todos os clientes tenham migrado para a nova API (ou passado a usar as bibliotecas atualizadas) e que os dados antigos estejam seguros no novo sistema, podemos decomissionar o antigo Log Center. O código legado (Flask) pode ser arquivado e seu serviço desligado. Os dados legados, se migrados, continuam disponíveis no novo Mongo; se por algum motivo optarmos por não migrar tudo (por volume ou relevância), poderíamos deixá-los num cluster Mongo legado apenas para consultas históricas, mas a preferência é migrar para ter tudo unificado.

Resumindo, a transição será feita de modo a **não interromper o fluxo de logs** e **não perder histórico**. A retrocompatibilidade na API garante que mesmo dispositivos não atualizados imediatamente continuem reportando, enquanto trabalhamos para atualizá-los conforme o guideline e novas bibliotecas.

9. Versionamento de API e Evolução do Esquema de Dados

Para garantir longevidade e facilidade de evolução do Log Center, definimos políticas de versionamento tanto da API quanto do esquema de dados:

- **Versionamento da API:** A API REST iniciará na versão `v1`. Todos os endpoints serão prefixados com `/api/v1/`. Qualquer mudança **incompatível** (break change) futura resultará na criação de uma nova versão (`v2`), mantendo a antiga disponível por um período de transição. Por exemplo, se decidirmos mudar o formato de autenticação ou remover campos obrigatórios, esses ajustes serão introduzidos em `/api/v2/logs` enquanto `/api/v1/logs` continua operando no modo antigo até ser descontinuado. Versões serão bem documentadas, e espera-se seguir versionamento semântico na medida do possível: *increases minor version for backwards-compatible additions, major version for breaking changes*. No contexto de rotas HTTP, isso significa usar o prefixo para versões principais apenas.

- **Evolução do Esquema de Dados:** Como os logs são armazenados em JSON (MongoDB), temos certa flexibilidade para adicionar novos campos sem afetar

registros antigos. Por exemplo, se futuramente quisermos incluir um campo `deviceId` ou `userId` em certos logs, podemos começar a enviar nos novos sem precisar reescrever todos os antigos – eles simplesmente ficarão sem esse campo (o sistema deve lidar com ausência de campos graficamente e na lógica). Contudo, remoção ou mudança de significado de campos requer cuidado:

- Evitar ao máximo **renomear ou remover campos** no esquema vigente. Em vez disso, pode-se **depreciar** campos antigos – parando de usá-los nos novos logs, mas mantendo-os no código por compatibilidade. Se realmente precisar remover, fazê-lo apenas em uma nova versão de API e, possivelmente, migrar dados antigos para o novo campo.

- Manter um **controle de versão do esquema**: podemos inserir um campo meta, como `schema_version` no documento, ou simplesmente documentar a data de mudança. Em APIs modernas com OpenAPI, a documentação já reflete o formato esperado por versão de API, então talvez não seja necessário embutir versão em cada documento, mas é algo a considerar para migrações automáticas (e.g., se abríssemos todos documentos para upgrade).

- **Pydantic Models Versionados:** Uma prática facilitadora é versionar os modelos Pydantic internamente. Por exemplo, termos `LogEntryV1` e no futuro `LogEntryV2`. Isso deixa explícito no código as mudanças e permite que endpoints antigos continuem usando o modelo antigo enquanto os novos usam o novo.

- **Compatibilidade das Bibliotecas Cliente:** Versões da API e do esquema impactam as libs. Faremos uso de versionamento de pacote (e.g., `logcenter-sdk-py` v1.x corresponde à API v1). Se sair a API v2, lançar `logcenter-sdk-py` v2.x que sabe lidar com as mudanças. Manteremos retrocompatibilidade sempre que possível nas libs (por exemplo, se apenas adicionamos um campo opcional, a lib antiga ainda funciona; se mudamos algo obrigatório, a lib detecta erro e talvez possa ajustar se trivial). A comunicação de mudanças será clara no changelog do projeto.

- **Deprecações:** Se um endpoint ou campo for deprecado, a documentação da API marcará isso e possivelmente o servidor emitirá avisos (ex.: header `Deprecation` nas respostas ou logs internos) quando notar uso de recursos antigos. Isso ajudará a identificar quem ainda está usando funcionalidades legadas e planejar sua atualização.

Em suma, adotaremos uma postura pró-ativa de versionamento para **permitir evolução sem prejudicar consumidores existentes**. Isso é crucial, pois o Log Center deverá durar e se adaptar a novas necessidades (novos tipos de log, integrações, etc.) ao longo do tempo, e um versionamento organizado evita **rupturas inesperadas**. A própria decisão de migrar para FastAPI já inaugura a **versão 1** de um API mais estável e padronizada, estabelecendo as bases para crescimento futuro.

10. Boas Práticas de Desenvolvimento e Escalabilidade

Para assegurar que o novo Log Center seja sustentável a longo prazo, enumeramos algumas diretrizes de boas práticas que complementam a especificação:

- **Código Limpo e Padronizado:** Seguir convenções de código Python (PEP8) e análogas nas libs JS/C#. Utilizar linters e formatação automática (black, eslint, etc.) para manter a qualidade. Comentários e docstrings nos pontos importantes (especialmente nas funções do serviço e nas bibliotecas públicas) para facilitar o entendimento. Evitar duplicação de código – fatorar funcionalidades comuns (por exemplo, geração de UUID, formatação de data) em funções utilitárias reutilizáveis.

- **Logging Interno do Sistema:** O próprio Log Center deve ter logs de atividade – por exemplo, registrar quando inicia, conexões de banco, erros em processamento de requests etc. Isso será feito utilizando a biblioteca `logging` do Python configurada apropriadamente (log para stdout/stderr no container, podendo ser coletado pelo Docker). Podemos definir um formato simples com timestamp, nível e mensagem para esses logs de sistema. Em produção, esses logs poderão ser agregados em uma solução central de observabilidade (ELK stack, por exemplo) se disponível.

- **Tratamento de Erros e Monitoramento:** Implementar manipuladores globais de exceções no FastAPI (via `exception_handler`) para capturar erros não previstos e retornar respostas HTTP coerentes (evitando que o servidor quebre silenciosamente). No painel, prever fallback se a API estiver fora (mostrar mensagem de erro amigável). Além disso, monitorar a performance – e.g., tempo de resposta médio das requisições de log – isso pode ser incorporado via middleware ou ferramentas APM. Como logs podem chegar em alta taxa, precisamos observar uso de CPU/memória e, se necessário, escalar horizontalmente ou otimizar pontos de contenção (por ex., inserções bulk no Mongo ao invés de uma por uma, se o throughput exigir).

- **Segurança:** Proteger contra inputs maliciosos – embora os logs venham de fontes conhecidas, é essencial validar os campos. Use das validações do Pydantic (e.g., tamanhos máximos para strings como `message` para evitar overflow, sanitização se for exibir em HTML no dashboard para prevenir XSS). Configurar corretamente o CORS no FastAPI para que apenas origens autorizadas (como o domínio do painel) possam consultar as APIs sensíveis, evitando exposição indevida. Se for aberto a Internet, considerar limites de rate (limitar, por IP ou por chave, quantos logs por segundo podem ser enviados, para prevenir abuso).

- **Escalabilidade e Distribuição:** Graças ao Docker/Kubernetes, o Log Center poderá facilmente escalar horizontalmente. FastAPI sendo stateless (com o estado no Mongo) permite rodar N instâncias atrás de um load balancer, atendendo maior carga de escrita/leitura de logs. O MongoDB, se necessário, também pode ser escalado (um cluster com replicação e sharding se o volume for muito grande). Planejamos desde já facilitar isso, por exemplo, mantendo configurações de conexão que suportem string de múltiplos hosts (no .env). O design de inserção de logs deve ser leve: inserir um documento no Mongo é O(1) e muito rápido, mas se começarmos a ter lógica extra pesada em cada log (como cálculo complexo ou chamadas a outros serviços), isso deve ser movimentado para processamento assíncrono para não degradar a ingestão. Podemos utilizar *background tasks* do FastAPI para qualquer pós-processamento não crítico na resposta imediata.

- **Testabilidade:** Escrever testes unitários para as partes críticas – e.g., teste do esquema (ver se Pydantic valida corretamente), teste dos endpoints (usando `TestClient` do FastAPI simulando envio de logs), teste das funções de agregação. Também testes para as libs clientes (em Python, podemos usar PyTest; em JS, frameworks de teste de JS; em C#, talvez usar NUnit) para garantir que formatação e envio estão ok. Isso evitará regressões quando o sistema evoluir (dado que esperamos várias iterações futuras).

- **Documentação e Guideline de Uso:** Além desta especificação, será produzido um **guia de uso** (para desenvolvedores integrando e para usuários do painel). Por exemplo, um manual rápido de "Como logar adequadamente": definindo o que deve ser logado ou não (evitar dados pessoais sensíveis nos logs, a não ser anonimizados, etc.), quais tags usar para certos cenários (padronização semântica), e dicas de desempenho (não chamar log em loop intenso, etc.). Inclusive, conforme acordado, Guilherme produzirá um guideline sobre como e o que logar, incluindo padrão de tags e uso de UUID[9] – estes pontos estão em linha com o que foi descrito aqui.

- **Manutenção de Dependências:** A stack escolhida (FastAPI/Python, libs JS, etc.) requer atenção a versões. Congele versões no requirements.txt/poetry lock para reprodutibilidade. Planeje janelas para atualizar dependências periodicamente, verificando breaking changes. Isso vem da lição do projeto Skyn, em que código antigo e bibliotecas desatualizadas causaram problemas. Com Docker, é fácil testar novas versões em ambiente isolado antes de promover.

Em conclusão, essas boas práticas garantirão que o Log Center reestruturado não apenas atenda aos requisitos funcionais imediatos, mas seja **confiável, seguro e escalável**. A equipe já se comprometeu a iniciar essa reestruturação imediatamente – Julio iniciando a codificação em FastAPI/Docker, Guilherme definindo estrutura de dados e documentação – e seguindo as diretrizes aqui estabelecidas, o novo Log Center se tornará um **sistema de log centralizado moderno** que suportará eficientemente os projetos atuais e o crescimento futuro.

Conclusão

Este documento apresentou uma análise crítica do Log Center atual, evidenciando seus principais problemas (falta de padronização, formatação inadequada de dados, decisões arquiteturais problemáticas) e contrastando-os com práticas modernas adotadas em projetos recentes. Com base nisso, delineamos uma especificação técnica completa para a reestruturação do Log Center, cobrindo desde a escolha de tecnologias (FastAPI, MongoDB, Docker) até detalhes de implementação (esquema JSON de logs com UUIDs e tags, bibliotecas cliente multi-linguagem, dashboard administrativo).

Seguindo esta especificação, o novo Log Center será capaz de: - Receber e gerenciar logs de múltiplas plataformas de forma consistente e eficiente; - Fornecer uma interface unificada para monitoramento e análise desses logs em tempo real; - Escalar conforme a demanda e evoluir sem sobressaltos graças a versionamento bem planejado e boas práticas de desenvolvimento.

Em suma, a reestruturação alinhará o Log Center com os padrões atuais de desenvolvimento de software e necessidades de observabilidade, transformando-o em um **serviço robusto de logging centralizado** e agregando valor tanto para os desenvolvedores quanto para os clientes que poderão extrair insights dos dados coletados. Conforme indicado nas reuniões, esta abordagem padronizada e moderna de logging era uma meta fundamental da equipe, e sua realização proporcionará uma base sólida para os futuros projetos da organização.