ChatGPT

Firestore Entity Model for Arcade AssetGen (Multi-Tenant Backend)

This data model uses a multi-tenant hierarchy in Cloud Firestore, with organizations at the top level and related data nested below. Each entity is detailed with its collection structure, key fields, relationships, and any denormalization or indexing strategies to meet the spec (e.g. storing prompt history, filtering by tags/themes, etc.). This structure aligns with Firebase best practices for multi-tenant apps [1] and optimizes for read-heavy access by duplicating data where necessary [2].

1. Organization

Collection Location: Top-level collection, e.g. `organizations/{orgId}` [1], where `{orgId}` is a unique ID (auto-generated or custom).

Fields:
- **name** – Organization name (string, used for display).
- **createdAt** – Timestamp of creation.
- **ownerUserId** – (Optional) ID of the user who created or owns the org (for admin purposes).
- **planTier** – (Optional) Org's subscription or plan info if relevant (for multi-tenant billing/limits).

Relationships:
- **Projects Subcollection:** Each org document contains a subcollection `projects` for projects under that organization (see Project entity below) [1].
- **Members Subcollection:** Each org can have a `members` subcollection (or similar, e.g. `users`) storing membership records. Each member document could include the **userId** and **role** (e.g. "admin", "editor", "viewer") within this org [1]. This allows listing users of an organization and role-based access control. Alternatively, an array of userIds/roles could be stored on the org document, but a subcollection is preferable to avoid large arrays in a single doc.
- **User Profiles:** A separate top-level `users` collection (see User entity) contains user profiles. Org membership docs link to these user profiles via userId, and conversely user profiles list orgs they belong to.

Denormalization & Indexing:
- **Org ID in Sub-documents:** The organization's ID is included in nested documents (like projects, assets) as a field for redundancy [1]. For example, each project or asset doc can include an `orgId` field. This duplication ensures that if data is queried outside the org path (e.g. via a collection group query), the client can filter by orgId and security rules can verify tenant isolation [3].
- **Member Info Duplication:** To optimize displays of org members, the member subcollection docs might store denormalized info such as the user's name or email (so the UI can show a member list without fetching each user profile). This is a safe duplication since user names change rarely.
- **Indexes:** Typically, org documents are fetched by ID (no complex query needed). If supporting lookup by name (e.g. searching org by name), index the `name` field (Firestore auto-indexes single fields by default). The members subcollection may be queried by role or userId; ensure those fields are indexed (automatic

for single field queries). For security, ensure queries are always scoped to the org (e.g. using the org path or an explicit orgId filter in composite indexes as needed).

2. Project

Collection Location: Subcollection under an organization, e.g. `organizations/{orgId}/projects/{projectId}` [1]. (Alternatively, projects could be a top-level collection with an `orgId` field, but nesting under org helps naturally partition data by tenant.)

Fields:
- **name** – Project name (string).
- **description** – (Optional) Description or summary of the project.
- **orgId** – Organization ID (string) that this project belongs to. *If projects are stored under* `organizations/{orgId}`*, this may be implicit from the path, but we include it for query support and redundancy* [1].
- **createdAt / updatedAt** – Timestamps for creation and last update.
- **settings** – (Optional) Project-level settings, such as default generation parameters or permissions (could be a map of config values).

Relationships:
- **Organization:** The project's parent is an Organization document (link via `orgId` or by location under the org). All assets, themes, etc. within this project inherit the org relationship.
- **Assets Subcollection:** Each project has an `assets` subcollection for asset documents (see Asset entity).
- **Themes Subcollection:** Each project has a `themes` subcollection for theme documents (see Theme entity).
- **ConceptImages Subcollection:** Each project has a `conceptImages` subcollection for reference images (see ConceptImage entity).
- **StyleGuide:** A project may have an associated style guide (markdown text). This can be stored either as a field in the project document (e.g. `styleGuideMarkdown`) or as a single document in a `styleGuide` subcollection (containing the markdown content). If using a subcollection, e.g. `projects/{projectId}/styleGuide/content`, the document would have fields like `markdownText` (and perhaps `lastEditedBy`). Project-level style guide provides overarching art direction for all assets in the project.

    - **Project Members:** If user access is managed at the project level (e.g. a user can belong to an org but only have access to certain projects), you can include a `projectMembers` subcollection under each project. Each member doc would contain `userId` and `role` for that project (similar to org members). This is useful for fine-grained access control when not all org users should see all projects. (In many cases, org-level membership implies access to all projects, so this is optional based on requirements.)

Denormalization & Indexing:
- **Org Data:** The `orgId` field on the project doc is indexed, allowing queries like "find all projects for Org X" if needed when using a top-level projects collection or collection group query [3]. In a nested structure, listing projects for an org is typically done by reading the subcollection directly, which is efficient.
- **Project Name for Search:** If supporting search by project name or ordering projects by name, Firestore's default single-field index on `name` suffices for simple queries. For compound queries (e.g. filter by orgId and sort by name in a flat structure), a composite index on (orgId, name) would be needed.
- **Denormalized Counts:** (Optional) The project document could maintain denormalized counts (e.g.

number of assets, number of themes) for quick display. These would be updated via Cloud Functions or transactionally when assets/themes are added. This avoids counting sub-documents on the fly in the client.

3. Asset

Collection Location: Subcollection under a project, e.g. `organizations/{orgId}/projects/{projectId}/assets/{assetId}`.

Fields:
- **name** – Asset name or key (e.g. `"hero_icon"`) – a human-friendly identifier for the asset.
- **description** – (Optional) Longer description of the asset or its purpose.
- **prompt** – The base prompt or description used for generation (text). This captures the core idea of the asset (e.g. *"A medieval hero character icon"*).
- **size** – Target image size or resolution for the asset. This could be stored as discrete fields (e.g. `width`, `height` integers) or a preset string (`"512x512"`) depending on how generation size is specified.
- **themeId** – Reference to a Theme used for this asset (if the asset is associated with a particular stylistic theme). This would typically store the theme's document ID.
- **conceptImageIds** – (Optional) An array of IDs of **ConceptImage** references used or associated with this asset. For example, if certain concept/reference images are always applied to this asset's prompt for inspiration, their IDs can be linked here. (The actual usage of concept images in each generation is also tracked per Generation record.)
- **tags** – (Optional) An array of tags or labels for the asset. This can be used to categorize assets (e.g. `["character","icon","medieval"]`). Tags make it easier to search or filter assets within a project by categories or features.
- **createdAt / updatedAt** – Timestamps for when the asset was first created and last modified.
- **createdBy** – (Optional) userId of the creator or owner of this asset.

Relationships:
- **Project:** The asset belongs to a single Project (the parent in the path). The field `projectId` can be included for redundancy if assets are ever queried outside the project context, but it's usually implicit from the path.
- **Theme:** If `themeId` is set, it references a Theme document in the same project. Consider storing a **reference** (`DocumentReference`) or simply the theme's ID string. This association means the asset's generations likely use that theme's style settings by default.
- **Concept Images:** If `conceptImageIds` are present, they refer to ConceptImage docs in the project's conceptImages collection. These images can be reused across multiple assets, so an asset can link to many concept images, and each concept image can be linked from many assets (many-to-many relationship). The link is maintained by storing the IDs here (and potentially in generation records when used). There is no direct foreign-key constraint, but integrity can be managed in the application logic (e.g. ensure an ID refers to an existing concept image).
- **Generations Subcollection:** Each asset has a `generations` subcollection, which stores the history of prompt runs for that asset (see Generation entity). Each generation doc represents one "version" of the asset's generated output (with possibly multiple variant images). This history is crucial for reproducibility – you can trace how the asset evolved over time.
- **Active/Final Variant:** (Optional) If it's useful, the asset document could have a field like `finalVariantId` or `currentImageUrl` pointing to the chosen final image (one of the Variants) that

represents the asset's current version. This avoids scanning the generations list to find the latest approved image each time.

Denormalization & Indexing:
- **Theme Data on Asset:** To avoid extra lookups when listing assets, you may denormalize some theme info on the asset. For example, store the theme's name or key attributes (like a short style label or color palette summary) in the asset doc. This way, when displaying a list of assets with their theme, you don't need to fetch each Theme doc separately. (If a theme is renamed, those asset docs should be updated to keep in sync – a manageable trade-off for faster reads [2].)
- **Concept Tags on Asset:** If filtering assets by concept image tags is needed, consider denormalizing concept image tags into the asset's own `tags`. For example, if an asset's linked concept images have a tag "sci-fi", copying "sci-fi" into the asset's tags array ensures that a query for assets tagged "sci-fi" will include assets that use such concept images. This avoids needing a complex join, since Firestore doesn't support joins and requires data to be duplicated for such queries [2]. (This denormalization should be maintained by updating asset tags if concept image tags change or if concept links are added/removed.)
- **Indexes for Filtering:** Firestore automatically indexes individual fields, but composite indexes are needed for multi-field queries. For example, to query assets by theme *and* tag within a project, you would create a composite index on `themeId` + `tags` (and `projectId` if using a flat structure) [4]. In our hierarchical model, the query to "filter assets by tag + theme in a given project" would be scoped to the `projects/{projectId}/assets` collection, with conditions `where("themeId","==",<themeId>)` and `where("tags","array-contains",<tag>)`. This requires a composite index on `themeId, tags` for the assets collection (Cloud Firestore will prompt for this index if it's not set automatically) [4].
- **Search by Name/Tag:** The `name` and `tags` fields are indexed by default for simple queries (e.g. `where("name" == "...")` or `array-contains` on tags). If you need to support partial text search on asset names or descriptions, consider integrating with a full-text search service (since Firestore only supports prefix queries via indexes, not full text search). But straightforward equality/contains queries are supported with proper indexes.

4. Generation

Collection Location: Subcollection under an asset, e.g. `.../assets/{assetId}/generations/{generationId}`.

Meaning: A Generation represents one run of the AI image generation for the asset. This includes the prompt configuration used and links to the output variants. Essentially, each generation is like a "version attempt" for the asset, and assets can have up to 1–100 generations (versions) according to the spec.

Fields:
- **promptText** – The full text prompt sent to the generation AI for this run. This should capture all components that were used (including the asset's base prompt plus any theme/style modifiers or concept influences applied). *Storing the exact promptText ensures reproducibility of the generation* [3].
- **parameters** – A map of generation parameters used (e.g. model settings, seed value, iteration steps, guidance scale, etc.). By logging these, you can re-run or tweak the generation later. For example, if using Stable Diffusion, this might include `modelVersion`, `seed`, `numSteps`, `cfgScale`, etc.
- **themeId** – The ID of the Theme applied during this generation (if any). This allows you to know which theme's style presets were used. You might also store a snapshot of critical theme settings here (e.g. the

theme's name or style keywords at the time) in case the theme is later modified – this ensures the generation record is self-contained for reproducibility.
- **conceptImageIds** – An array of ConceptImage IDs that were applied in this generation (if any). This records which reference images influenced the AI output for this run. (For example, if the user attached a "spaceship_reference" image to guide this generation, its ID appears here.) As with theme, you might store some info about each concept image's effect (like weights or prompts, if used) or at least ensure the concept images remain available for reference.
- **variantCount** – Number of variants (images) produced in this run (integer). If the generation created multiple images, this notes how many.
- **createdAt** – Timestamp when this generation was executed. This can serve as the version timestamp.
- **triggeredBy** – The user ID who initiated the generation (to track who made a version).
- **status** – (Optional) Status of the generation process, e.g. `"completed"`, `"failed"`, or `"pending"` if generation is asynchronous. In a serverless setup, you might create the generation doc with status pending, then a Cloud Function updates it to completed and adds variants.
- **notes** – (Optional) Any user-provided notes about this generation (e.g. "used a high CFG for sharper details").

Relationships:
- **Asset:** The generation belongs to an Asset (parent path contains assetId). It inherits `projectId` and `orgId` context through the hierarchy. For safety or convenience, you could store `assetId` (and even project/org ID) on the generation doc, but these are usually known from the path. Storing them can help if doing any collection group queries on generations across assets or projects (then you'd filter by projectId and/or orgId) [3].
- **Variants Subcollection:** Under each generation, there is a `variants` subcollection containing the actual image outputs (see Variant entity). Each variant doc holds one image and its metadata.
- **Theme/Concept References:** The `themeId` links to a Theme doc, and `conceptImageIds` link to ConceptImage docs (both in the same project). These references connect a generation to the styling inputs used. No direct foreign-key enforcement exists, but application logic or security rules can ensure these IDs belong to the same project and org for consistency.
- **Retrieval:** To get an asset's full prompt history, you query its generations subcollection (optionally sorted by createdAt or a version number). Each generation doc is essentially a log entry that, combined with its variants, fully describes one version attempt.

Denormalization & Indexing:
- **Self-Contained Prompt Info:** We duplicate relevant info in the generation record so that it's self-contained. For instance, even though the asset has a base prompt and theme, we store the exact `promptText` used after combining the base prompt, theme keywords, style guide cues, and concept influences. This way, the generation can be understood on its own later, and if the theme or asset prompt changes afterwards, the historical record remains intact. This is a form of denormalization used to ensure reproducibility and traceability [2]. Similarly, storing `themeId` and `conceptImageIds` (and possibly their names/tags at that time) in the generation doc duplicates some data, but it **guarantees reproducibility** – you know exactly what inputs produced a given variant.
- **Version Number:** If you prefer explicit versioning, you could include a field like `versionNumber` (1, 2, 3, ...) on each generation. This could simply be an auto-increment per asset. However, since Firestore can't auto-increment safely, this would be set by the client or a transaction that reads the current count and increments. Many apps just use timestamp ordering instead of explicit version numbers. If you do store a version number, consider indexing it if you frequently query by version.

- **Querying Generations:** Typically generations are fetched in context of an asset (e.g. “show me all versions of asset X”). This is a simple collection query within `assets/X/generations`. If needed, you can filter or sort, e.g. order by `createdAt desc` to get latest first. Default indexing on `createdAt` supports ordering by timestamp. If you query generations by other fields (say find all generations by a certain user, or all that used a certain theme across the project), you might use a collection group query on `generations` across all assets. In that case, ensure to include `projectId` (and `orgId`) in generation docs and use them in the query filter [3]. For example, a collection group query could target all `generations` documents where `projectId == "123"` and `themeId == "neo-noir-theme"`. A composite index on (`projectId`, `themeId`) would be required to support that multi-field query across the collection group.
- **Cleanup/Retention:** (Not exactly indexing, but important) If prompt history needs to be retained indefinitely for reproducibility, there’s no issue. If there are concerns about old versions, you could implement a policy to archive or delete old generation docs after a certain limit (100 per asset as given) to keep the dataset lean, or move them to cold storage. Firestore supports TTL indexing if needed for automated cleanup of old docs.

5. Variant

Collection Location: Subcollection under a generation, e.g. `.../generations/{genId}/variants/{variantId}`.

Meaning: A Variant represents one image output from a generation run. If a generation produces multiple images (say 4 variants for the prompt), each is stored as a Variant document. This separation allows storing metadata per image and tracking user feedback or selection of each variant.

Fields:
- **imageUrl** – URL or storage path to the variant image (e.g. a Cloud Storage URI or download URL). This is the core output – the actual image file is stored in Firebase Storage or a similar service, and this field holds the reference.
- **thumbnailUrl** – (Optional) URL/path for a thumbnail or lower-res preview of the image, if you store those for quicker loading in the UI.
- **metadata** – (Optional) Any metadata for this image variant. This could include generation parameters specific to this image if the generation API provides per-image info (for example, some diffusion models might return a seed used for each variant if they vary per image). Typically, if all variants share the same settings except random seed, you might store the seed here if each image had a different seed.
- **isSelected** – (Optional) Boolean or enum to mark if this variant was selected or approved by the user. For example, the user might pick one variant as the final asset image; that variant’s doc could have `isSelected: true` (and others false). Alternatively, a separate field on the Asset can reference the chosen variant’s ID, as mentioned earlier.
- **feedback** – (Optional) User feedback or rating for this variant (e.g. a numeric rating, or a comment like “too dark”/“good detail”). This can help in iterating on the prompt or just record why a variant was or wasn’t chosen.

Relationships:
- **Generation:** The variant’s parent is a Generation document. It inherently links back to an Asset (and project/org) via that path. The variant doc can include an `generationId` or `assetId` field for

completeness, but usually the parent reference is enough. If you ever needed to query variants without knowing the generation (rare in this model), including those IDs would be necessary to filter correctly.
- **Asset (indirect):** Indirectly, each variant belongs to a specific asset. If needed for security or querying, you can carry the `projectId` and `orgId` fields here as well (ensuring any collection group query on variants can enforce tenant boundaries [3]). However, this is often not required since you will fetch variants via known generation/asset paths.
- **No separate sub-relations** typically under a variant (variants are leaf nodes containing the image info). If variant images themselves had further detail (say segmentation or metadata), they could be stored in the same doc or a subcollection if complex, but that's likely unnecessary.

Denormalization & Indexing:
- **Minimal Denormalization:** Variants are usually retrieved in context, so little duplication is needed. The variant doc itself is already small (mostly a pointer to storage and a couple fields).
- **Indexing:** Each field in the variant doc (imageUrl, isSelected, etc.) is automatically indexable singly. In most cases, you won't be running complex queries on variants alone – you retrieve them by generation. If you did need to query across all variants (e.g. "find all selected variants in project X" for some analytics), a collection group query on `variants` with filters (like `projectId == X` and `isSelected == true`) would require those fields in the variant docs and a composite index on (projectId, isSelected). This is only if such cross-generation queries are needed, which is uncommon in normal operation.
- **Storage Consideration:** The actual images are not stored in Firestore, only references, so no index bloat from image data. Ensure your filenames or storage paths encode the necessary info (like org/project for easy lookup in storage). Storing the download URL or a path string in Firestore is fine (string fields are indexed by default, but you could exempt large URL fields from indexing if needed to save space/cost, since you rarely query by the URL itself).
- **Deletion:** When an asset or generation is deleted, variants should be deleted as well (Firestore can't automatically cascade delete subcollections, so use a Cloud Function or client logic to clean up, or use Firestore's recursive delete function).

6. Theme

Collection Location: Subcollection under a project, e.g. `.../projects/{projectId}/themes/{themeId}`.

Meaning: A Theme represents a reusable style preset (palette and keywords) that can be applied to assets. Themes are defined per project but can be exported/imported to other projects (meaning the data model should allow themes to be easily copied or recreated in another project).

Fields:
- **name** – Theme name (string), e.g. `"Noir Comic Style"` or `"8-bit Pixel Art"`.
- **styleKeywords** – A list of keywords or prompt snippets that define the style. For example: `["high contrast","film noir","grainy texture"]` or `["8-bit","pixelated","arcade style"]`. These are appended or integrated into prompts when the theme is used.
- **colorPalette** – A representation of the color palette for the theme. This could be an array of color hex codes (`["#000000","#FFFFFF", ...]`) or a reference to a palette object. If the palette is complex (e.g. with metadata like name of each color), it could be a map or even a subcollection of colors, but typically a simple list of hex values or a JSON blob of palette info suffices.

- **description** – (Optional) A textual description of the theme's mood/usage, e.g. "Use for dark, mysterious elements and dramatic shadows."
- **exampleImage** – (Optional) a URL or reference to an image that exemplifies the theme's style (could be one of the concept images or an external image).
- **projectId** – The ID of the project this theme belongs to (redundant if under project path, but useful if themes are ever queried in aggregate or moved).
- **createdBy** – (Optional) userId who created the theme.
- **createdAt / updatedAt** – Timestamps for creation and last modification.
- **styleGuideMarkdown** – (Optional) A field for any extended style guide text specific to this theme (if the theme has its own guidelines beyond the raw keywords and palette). This is essentially a theme-specific **StyleGuide**. If substantial, this could be stored as a separate document (e.g. a subcollection `styleGuide` under the theme), but usually including a `styleGuideMarkdown` field here is sufficient.

Relationships:
- **Project:** The theme belongs to a project (by path and `projectId`). All assets in the project can potentially use this theme.
- **Assets:** There's no direct list of assets in the theme document, but assets reference themes by themeId. To find all assets using a given theme, one would query the project's assets where `themeId == this themeId` (this is straightforward with an index on asset.themeId). This is one reason to keep theme and asset in the same project partition.
- **Concept Images:** Themes can be linked to concept images as well. For example, a theme might have a set of reference images illustrating its style. To model this, you could add a field `conceptImageIds` (array) in the theme doc, listing concept images that belong to or exemplify the theme. (E.g. a "Noir" theme might link to a few black-and-white movie stills stored as concept images). This is optional, but it aligns with *"concept images … can be reused across assets or linked to themes."* If used, these concept images would still reside in the project's conceptImages collection, and their IDs are referenced here (and possibly a boolean flag on the concept image like `forTheme: themeId` if needed).
- **Style Guide:** If the project has a global style guide, and the theme has a specific one, the theme's own guide (if any) is stored in `styleGuideMarkdown` or in a sub-doc. The relationship is one-to-one: each theme could have its own extended guidelines.
- **Export/Import:** When exporting a theme, all these fields define the exportable data. Importing would create a new theme doc (with a new themeId in the target project) containing the same fields. There's no global theme registry in Firestore by default; it's usually handled via JSON export or copy. (If needed, one could implement a top-level `themes` collection for templates that can be copied into projects, but that's beyond the core scope.)

Denormalization & Indexing:
- **Project Info:** The `projectId` (and possibly `orgId`) on the theme doc ensures that if we do any cross-project theme queries or use collection group queries (unlikely, since themes are usually just listed per project), we can filter by the correct project/organization [3]. In normal operation, you'd list themes by simply reading the `projects/{id}/themes` subcollection, which is already scoped.
- **Theme Name Usage:** Within an asset or generation, we often store the `themeId`. If needed for display or filtering, we might also store the `themeName` on related docs (asset or generation). For example, in the Asset entity we mentioned denormalizing the theme name for quick listing. This is duplication but improves performance when showing assets with their theme names without an extra lookup [2]. Keep in mind if a theme name changes, asset docs should be updated for consistency.
- **Indexes:** Firestore will auto-index the theme's fields for simple querying. Common operations: listing all

themes (just get all in subcollection), or querying by name (you can use `.where("name", "==", "Noir")` which is fine with a single-field index). If you allow filtering themes by certain attributes (say find themes created by a user, or themes that include a certain keyword in their styleKeywords), you might create indexes: - Composite index on `(createdBy, name)` if you list themes by creator and sort by name. - Array-contains index on `styleKeywords` is automatic for single term search; if you need to filter themes that contain a specific keyword, Firestore supports `array-contains "keyword"`. For multiple keywords (AND queries), you might need multiple queries or a composite index if combining filters. Usually one keyword at a time is manageable.
- **Denormalizing Concept Info:** If concept images are linked, you might store a bit of info about them (like their URLs or descriptions) in the theme for an export. However, since concept images are a separate entity, it might be cleaner to export them separately. In use, to display a theme with its example images, you'd query the conceptImages by the list of IDs. If you anticipate showing theme + concept thumbnails frequently, you could store a small thumbnail URL array in the theme doc (duplication for convenience).

7. ConceptImage

Collection Location: Subcollection under a project, e.g. `.../projects/{projectId}/conceptImages/{imageId}`.

Meaning: A ConceptImage is an image used as a reference or inspiration. These could be style reference images, mood boards, or specific concept art that can be attached to generation prompts. They are stored per project (each project has its own library of concept images) but can be reused across many assets and themes within that project. They are not tied to a single generation; rather, they serve as reusable building blocks.

Fields:
- **imageUrl** – URL or storage path of the concept image (where the image file is stored). Like variants, this points to Cloud Storage or similar.
- **thumbnailUrl** – (Optional) URL/path for a thumbnail of the concept image for quick display.
- **tags** – (Optional) Array of tags/keywords describing the concept image (e.g. `["character","robot","sci-fi"]`). These tags help in searching the concept library (e.g. find all concept images tagged "robot") and can be used for filtering assets indirectly if assets use those images.
- **description** – (Optional) A text description or title for the image (e.g. "Spaceship interior concept art"). This is helpful for human reference.
- **attribution** – (Optional) If the image comes from a specific source or has credit info, store it here.
- **uploadedBy** – (Optional) userId who added the image.
- **createdAt** – Timestamp when added.
- **themeId** – (Optional) If this image is primarily associated with a Theme, you can store the theme ID here. For example, if an image was imported as part of a theme's style reference, this links it. In practice, a concept image could be used in multiple themes, but if it's explicitly part of one theme's definition, this field can note that. (It's not a hard link – it's more for organization or filtering: e.g. you could filter concept images by themeId to see all images associated with a particular theme's style.)
- **assetId** – (Optional) If this image was uploaded specifically for a particular asset's inspiration, you might note that here. However, since concept images are meant to be reusable, usually you'd leave this null and just link via references when used. An image could be initially for one asset but later repurposed for others.
- **usageCount** – (Optional) An integer count of how many times this image has been used in generations (or in how many assets/themes it's linked). This can be updated by Cloud Function or transaction each time the

image is used. It's not critical, but could be helpful for analytics or for cleaning up unused images (if `usageCount` remains 0, the image might be safe to delete).
- **projectId** – The ID of the project (redundant if under project, but included for consistency and for potential collection group queries).

Relationships:
- **Project:** The concept image belongs to a specific project (no cross-project sharing unless you explicitly copy it to another project's conceptImages).
- **Assets:** There is no direct list of related assets on the concept image doc, but assets store references to conceptImageIds. To find which assets use a given concept image, you would query the assets collection for that project with `array-contains <imageId>` on their `conceptImageIds` field. This requires that we denormalize the concept usage into the asset (which we did include in Asset entity). If this becomes a common query, it's supported by an index on that field.
- **Themes:** Similarly, themes may reference concept images (via their `conceptImageIds` list). To find if a concept image is used by a theme, you could query themes where `conceptImageIds` contains the image's ID. Alternatively, use the `themeId` field on the image if it's singly associated. If concept images are heavily reused, it might not be worth tracking all usages in each image document (that would be a many-to-many mapping). Instead, rely on queries on the other side (assets/themes).
- **Generations:** Each generation that uses a concept image will list its ID. There is no direct reference from the concept image doc to generations (which would be impractical to maintain). If needed for analysis, a collection group query on generations could find all generations that used a particular concept image ID. (This is an advanced use – typically not needed in normal app flow.)

Denormalization & Indexing:
- **Tag Indexing:** The `tags` array on concept images is indexed for membership queries. You can query conceptImages by tag (e.g. find all concept images with tag "sci-fi" via `.where("tags", "array-contains", "sci-fi")`). Firestore automatically supports array-contains with a single-field index. If you want to filter by tag *and* something else (say themeId), a composite index on (themeId, tags) would be needed.
- **Name/Desc Search:** If you allow searching concept images by description or name text, Firestore's querying is limited (you'd need a full-text search service for arbitrary text). For exact matches or simple prefixes, you could index the `description` field. Given the scope, tags are the preferred way to categorize for search.
- **Project Field:** Including `projectId` (and `orgId`) on each concept image doc is useful for sanity checks and potential collection group queries. For example, an admin tool might query all concept images across all projects in an org; having orgId on each doc allows filtering the collection group by orgId [3]. Security rules can also use this to ensure a user from one org cannot access concept images from another (though path-based rules can handle it as well).
- **Denormalization into Assets/Themes:** As noted, we copy concept image references into asset and theme docs (lists of IDs). This duplication is by design for flexibility and filtering. It means when a concept image is deleted or renamed, you might need to update or remove those references in assets/themes. Typically, if a concept image is deleted, you'd run a cleanup to remove its ID from any `conceptImageIds` fields in the project.
- **Storage and Size:** As with variants, the actual image binary is not in Firestore. Storing many concept image metadata docs (a few thousand is fine) is cheap, and listing or filtering them by tags is efficient with proper indexes. Just avoid storing extremely large fields (like base64 data) in these docs; use Storage for the images.

8. StyleGuide

Collection Location: *Varies – tied to Project or Theme.* There isn't a free-floating styleGuide collection; instead the style guide content is associated with either a project or a theme, as described below.

Meaning: A StyleGuide is a markdown text document containing guidelines, references, or instructions about the art style. This could be an overall guide for a project or a specific guide for a theme. It's essentially documentation that informs prompt creation and design consistency.

Storage Approaches:
- **Project StyleGuide:** If the entire project has a style guide, you can store it as a field in the Project document (e.g. `styleGuideMarkdown` containing the markdown text). If the text is very large (close to the Firestore 1MB per document limit), or if you want to edit it independently, you could use a subcollection: e.g. `projects/{projectId}/styleGuide/{docId}` with a field `markdownText`. Since typically one style guide per project exists, keeping it in the project document for simplicity is common (and you can exempt that field from indexing to avoid overhead, as it's not used in queries [2]).
- **Theme StyleGuide:** For theme-specific style notes, you can include a `styleGuideMarkdown` field in the Theme document. This keeps the theme and its guide together. If preferred, a subcollection under the theme (e.g. `themes/{themeId}/styleGuide/{docId}`) could store the text, but that's usually unnecessary unless the theme's guide is extremely long or requires versioning.
- **Versioning:** If maintaining revision history of style guides is important, a subcollection could store multiple dated versions of the guide (each as a doc). Otherwise, a single field updated in place is fine.

Fields (for a style guide entry):
- **markdownText** – The full markdown content (string). This can include formatting, links to concept images, etc.
- **lastEditedBy** – (Optional) userId of who last edited the guide.
- **lastEditedAt** – Timestamp of last edit.
- **title** – (Optional) a title for the style guide document (e.g. "Project Art Direction Guide" or "Noir Theme Guidelines"), if needed.

Relationships:
- **Project:** A project-level style guide lives in or under the project doc. It applies broadly to all assets in the project. Everyone with access to the project can typically read this.
- **Theme:** A theme style guide is part of the theme entity. It refines how to apply that theme. For example, the theme "Pixel Art" might have a style guide section explaining how to draw outlines or shading consistently.
- **No direct subentities** under a style guide (unless you wanted something like comments or revisions, but that's outside our scope).
- If style guides need to be exported/imported, they would be included when transferring a theme or project (just copy the markdown text accordingly).

Denormalization & Indexing:
- **No Need for Duplication:** We generally do not duplicate the style guide text anywhere else – other entities might reference it (e.g. an asset might refer to project's style guide implicitly by using theme), but they don't copy the text. The style guide is typically read as needed rather than queried against.

- **Indexing:** It's recommended to **disable indexing** on the markdown text field, because it's large and not used for queries [5] [2]. This can be done by setting the field to be exempt from indexing in Firestore rules or via the console. We do not need to run queries on the content of a style guide (no equality or range checks on large text), so indexing it only wastes storage and slows writes.
- **Access:** Since styleGuide is tied to project/theme, the same security rules for those entities apply (e.g. only org members can read the project's styleGuide). Keeping it within the project/theme document or as a subcollection ensures it's under the same security scope.
- **Retrieval:** To fetch a style guide, one would either get the project doc (to read the field) or get the theme doc. If stored in a subcollection (say `styleGuide/{doc}`), that's one extra query – which may be justified if the content is large (so you only load it when needed, not every time you load the project/theme). The decision comes down to frequency of access vs. data size. For simplicity and given moderate sizes, storing in the main doc is fine.

9. User

Collection Location: Top-level collection, e.g. `users/{userId}` for user profiles. (This is separate from Firebase Authentication's internal user store; here we store profile and authorization info.)

Fields:
- **name** – User's display name.
- **email** – User's email address (for reference; note that Auth also has this, but duplicating in profile is convenient for admin or queries).
- **profilePictureUrl** – (Optional) URL to avatar image.
- **orgMemberships** – An array or map listing the organizations and roles this user belongs to. There are a few ways to model this: - As an **array of objects**: e.g. `orgMemberships: [ { orgId: "org123", role: "admin" }, { orgId: "org456", role: "viewer" } ]`. This directly lists each org and the user's role in it. - Or as a **map**: e.g. `orgRoles: { "org123": "admin", "org456": "viewer" }`. This is similar, just keyed by org ID.
If the user can have project-specific roles, you might include those too (see below).
- **projectMemberships** – (Optional) If a user can be invited to specific projects (not all projects of an org), you might include a structure for that. For example, `projectMemberships: [ { projectId: "proj789", role: "contributor" } ]`. Alternatively, handle this by having the project's own member subcollection as mentioned in Project entity. Storing it here as well can be a convenient lookup of "which projects can this user access?". This can be denormalized because that info also exists in project member lists.
- **lastLogin** – (Optional) timestamp of last login (could be updated via Cloud Function on auth).
- **preferences** – (Optional) user-specific preferences/settings for the app (not directly relevant to data model relationships, but common in a profile document).

Relationships:
- **Organizations:** A user can belong to multiple organizations. The `orgMemberships` field (or subcollection) captures this. Additionally, each Organization's `members` subcollection contains a doc for the user (with role, etc.), effectively mirroring this relationship from the org side. This duplication means when a user is added to an org, we create an entry in both places (org's members and user's orgMemberships) for easy lookups from either direction.
- **Projects:** If using project-level access control, a user might not have full org access but only specific projects. In that case, either: - The user is still listed in the org's members (possibly with a base role like

“project-contributor”) and then project-specific roles are managed in project member lists. - Or we don’t list them in org members until they have at least one project; but usually if they’re in any project, they’re effectively part of the org (perhaps with limited scope).
In the data model, the project’s `projectMembers` subcollection would have an entry for the user. You can also reflect that in the user profile by listing those projects in `projectMemberships`. This helps to quickly answer “what can this user access?” without scanning all orgs and projects.
- **Security & Roles:** The roles in orgMemberships could determine what the user is allowed to do (e.g. an “admin” may manage all projects in that org, invite users, etc., whereas a “viewer” might only view generated assets). Project roles might be things like “contributor” vs “viewer” specific to one project. The data model just stores these; enforcement is via security rules or backend logic.
- **Auth Integration:** Typically the Firestore user doc ID will match the Firebase Auth user UID for consistency. The user doc acts as an extension of Auth, storing additional info and links (like org IDs). When a user signs up or is invited, you’d create this profile doc.

Denormalization & Indexing:
- **Org/Project Lists:** We deliberately duplicate membership info in both user and org documents for flexibility [2]. For example, to list all users in an organization, you read `organizations/{id}/members`; to list all orgs a user is in, you read the array/map in `users/{id}`. This avoids expensive queries like “find users where orgMemberships contains X” which, while possible with array-contains, would not scale well for large user base. Instead, we maintain the inverse relationship explicitly.
- **Indexing for Memberships:** If you use an array of membership objects, Firestore can’t directly index inside objects easily. If you use a map of orgRoles, you can query for existence of a key (not directly, Firestore doesn’t allow querying map keys without knowing them). So generally, we don’t query this field except to fetch the whole list for a given user. For the inverse (all users in an org), the org’s subcollection is used. That means no complex index needed on user’s memberships.
- **Email Index:** If you need to lookup a user by email (for login invitations or admin lookup), ensure the `email` field is indexed (it is by default for equality queries). You can do `users` collection query `.where("email", "==", "user@example.com")`. This might need a composite index if combined with other conditions, but usually not.
- **Name Search:** Similar to email, you can query by exact display name. For partial name search or case-insensitive search, consider using a search service. Sometimes apps add a lowercase version of the name as `name_lower` field and index that to do case-insensitive matches. This is a form of denormalization for search convenience.
- **Security Consideration:** Storing org and project IDs in the user doc also allows security rules to easily check if `request.auth.uid` has that org in their profile when trying to access an org’s data, as a secondary enforcement. The primary should be org’s member list, but having it on the user can be convenient for rule writing. (One must be careful to keep them in sync).

Indexing Summary: Given the above schema, most queries are naturally bounded by the hierarchy (org -> project -> asset -> generation), which is efficient. However, for composite queries (multiple filters), we plan and create composite indexes. For example: filtering assets by project + theme + tag requires a composite index on those fields [4]; querying concept images by project + tag needs one on projectId+tags; etc. We also selectively **denormalize data** (duplicate fields) to enable queries that would otherwise be impossible or too slow in Firestore’s NoSQL model (since we can’t join tables) [2]. This denormalization is used judiciously – for instance, storing `orgId` in deep entities for security filtering [3], or copying theme names into assets

for fast lookups. It's a common practice in Firestore data modeling to trade extra writes/storage for simplified reads and queries [2].

By following this entity design, the Arcade AssetGen backend can scale to multiple organizations and projects, store rich history for reproducibility, and support efficient queries (with appropriate indexes) for use cases like filtering assets by tags or themes within a project. Each entity is structured to keep data isolated per tenant while still allowing reuse and referencing of shared resources like themes and concept images across assets in the same project. The result is a flexible yet organized Firestore schema tailored for the asset generation workflow in a multi-tenant environment.

Sources:

- Firebase recommendation on multi-tenant data partitioning (organizations with subcollections for contained data) [1].
- Firestore does not filter by security rules in collection group queries, so including a tenant (org) ID in each doc and using it in queries is crucial for multi-tenant security [3].
- Denormalization (duplicating data such as names or IDs across documents) is a normal practice in NoSQL databases like Firestore to optimize read queries [2].
- Firestore requires composite indexes for querying multiple fields (e.g. filtering by theme and tag together) – each unique compound query must have an index configured [4].

---

[1] How to create a multi-tenancy structure using data silo for organizations and their users? - Database - Google Developer forums
https://discuss.google.dev/t/how-to-create-a-multi-tenancy-structure-using-data-silo-for-organizations-and-their-users/101209

[2] What is denormalization in Firebase Cloud Firestore? - Stack Overflow
https://stackoverflow.com/questions/54258303/what-is-denormalization-in-firebase-cloud-firestore

[3] firebase - What is the best practice for a multi-tenant app with Collection Group Queries in Firestore? - Stack Overflow
https://stackoverflow.com/questions/56410515/what-is-the-best-practice-for-a-multi-tenant-app-with-collection-group-queries-i

[4] java - Firestore with complex filter and composite indexes - Stack Overflow
https://stackoverflow.com/questions/62993392/firestore-with-complex-filter-and-composite-indexes

[5] Best practices for Cloud Firestore | Firebase
https://firebase.google.com/docs/firestore/best-practices