An Enterprise Architect's Playbook: Launching a GenAI MVP on Vertex AI in 7 Days

This playbook provides a strategic, day-by-day guide for rapidly deploying a Generative AI Minimum Viable Product (MVP) on Google Cloud's Vertex AI platform. The approach is engineered for both speed and discipline, establishing a foundation that is secure, scalable, and financially governable from the outset. By adhering to enterprise-grade architectural principles, this guide addresses the complete lifecycle of a Proof of Concept (POC)—from automated infrastructure provisioning and application development to rigorous cost tracking and long-term operational optimization. The objective is to not only ship a functional application within a seven-day sprint but also to establish a clear and sustainable path to ultra-low-cost operation after exhausting the initial free credits.

Part 1: Foundational Infrastructure with Terraform (Days 1-2)

The initial phase of any robust cloud deployment is the establishment of a solid foundation. Using Infrastructure-as-Code (IaC) is a non-negotiable practice for creating a repeatable, version-controlled, and secure environment. This approach is paramount for a POC, as it ensures that the entire setup can be audited, reproduced, or cleanly destroyed without manual intervention or configuration drift.[1]

1.1 Architecting for a Lean and Secure POC

The guiding principle for this architecture is the Principle of Least Privilege. The application's identity and its associated permissions will be scoped to the absolute minimum required for it to function, thereby reducing the potential attack surface. Terraform, an IaC tool from HashiCorp, will be used to declare the desired state of the Google Cloud infrastructure. This declarative syntax allows for the definition of

resources without specifying the procedural steps to create them, which Terraform handles by interacting with the Google Cloud APIs.[2]

Before proceeding, the local development environment must be prepared by installing the Terraform CLI and configuring the Google Cloud provider. This is typically achieved by first installing and initializing the Google Cloud CLI (gcloud) and then authenticating via a web flow, which generates local Application Default Credentials (ADC) that Terraform can use.[3]

1.2 The Core Infrastructure Module: A Production-Ready Snippet

The following Terraform configuration defines the three essential and interdependent resources required to enable Vertex AI and grant access to an application. This single, cohesive snippet fulfills the primary infrastructure requirement.

Terraform

```
# main.tf

# Configure the Google Cloud provider
provider "google" {
  project = "your-gcp-project-id" # Replace with your Google Cloud Project ID
  region  = "us-central1"
}

# 1. Enable the Vertex AI API for the project.
# This is the master switch that makes all Vertex AI features available.
resource "google_project_service" "vertex_ai_api" {
  project = "your-gcp-project-id" # Replace with your Google Cloud Project ID
  service = "aiplatform.googleapis.com"

  # Setting disable_on_destroy to false is a best practice. It prevents
  # Terraform from disabling the API if you destroy these specific
  # resources, which could impact other services in the project.
```

```
  disable_on_destroy = false
}

# 2. Create a dedicated service account for the application.
# This isolates the application's identity and permissions, adhering
# to the Principle of Least Privilege.
resource "google_service_account" "poc_sa" {
  project      = "your-gcp-project-id" # Replace with your Google Cloud Project ID
  account_id   = "sa-universal-clone-poc"
  display_name = "Service Account for Universal Clone POC"
  description  = "Grants access to Vertex AI models for the POC application."

  # This resource depends implicitly on the project existing, but not
  # on the API being enabled. However, the next resource depends on both.
}

# 3. Grant the service account the necessary permissions to use Vertex AI.
# The 'roles/aiplatform.user' role is a predefined role that grants
# permissions to make prediction calls and manage AI resources.
resource "google_project_iam_member" "poc_sa_vertex_user" {
  project = "your-gcp-project-id" # Replace with your Google Cloud Project ID
  role    = "roles/aiplatform.user"

  # This interpolation creates an implicit dependency. Terraform understands
  # that it must create the service account before it can grant a role
  # to that account's email address.
  member  = "serviceAccount:${google_service_account.poc_sa.email}"
}
```

Resource 1: Enabling the Vertex AI API
The google_project_service resource manages API services for a project.[4] The block above enables
aiplatform.googleapis.com, which is the endpoint for all Vertex AI services. The argument disable_on_destroy = false is a critical safety measure; it ensures that if this Terraform configuration is destroyed, the API remains enabled, preventing unintended disruption to other applications or services within the same project that might also rely on Vertex AI.[5]

Resource 2: Creating the Application's Identity
The google_service_account resource creates a new, non-human identity for the application.[6]

Assigning a unique service account (
sa-universal-clone-poc) is a foundational security practice. It allows for granular permission management and auditing, isolating the application's access from that of human users or other services. The account_id must be unique within the project and follows specific naming conventions.[7]

Resource 3: Granting Essential Permissions
The google_project_iam_member resource is used to attach an Identity and Access Management (IAM) policy to the project.8 This specific block binds the newly created service account to the predefined
roles/aiplatform.user role. This role contains the necessary permissions, such as aiplatform.endpoints.predict, to allow the service account to make prediction requests to Vertex AI models.[9] The choice of

google_project_iam_member is deliberate; it is a non-authoritative resource, meaning it adds a single member to a role without affecting other existing members of that role. This makes it safer to apply in projects with pre-existing IAM configurations, unlike the authoritative google_project_iam_binding which would overwrite all existing members for that role.[8]

1.3 The Critical Infrastructure Dependency Chain

A frequent point of failure in cloud deployments is a permissions mismatch, often manifesting as an opaque 403 Permission Denied error. The three Terraform resources above are not merely a checklist but form a critical dependency chain that, when automated, prevents such errors.

The logical sequence of operations is as follows:

1. The aiplatform.googleapis.com service must be enabled first. Without this, Google Cloud has no concept of Vertex AI-specific resources or the IAM roles associated with them, like roles/aiplatform.user.
2. The service account, the principal or "who," must be created. This provides the identity that will be granted permissions.
3. Finally, the IAM policy must be applied to connect the principal (the service account) to the permission (the roles/aiplatform.user role), defining "what it can do."

Terraform manages this sequence elegantly through resource interpolation. The line member = "serviceAccount:${google_service_account.poc_sa.email}" in the google_project_iam_member resource tells Terraform that it needs the email attribute from the google_service_account.poc_sa resource. This creates an implicit dependency, forcing Terraform to fully create the service account before it attempts to create the IAM binding. This built-in dependency management ensures the resources are provisioned in the correct order, handling potential eventual consistency delays in the underlying Google Cloud APIs and creating a more robust and reliable IaC configuration.[7]

Part 2: The AI Application Core in Python (Days 3-4)

With a secure and automated infrastructure foundation in place, the focus shifts to developing the Python application logic. This core component will be responsible for authenticating with Google Cloud and interacting with the text-bison and Gemini models on Vertex AI.

2.1 Environment Setup and Authentication

The development environment requires Python 3.11 and the Google Cloud AI Platform SDK. These can be set up within a virtual environment to isolate dependencies.

1. **Create and activate a virtual environment:**
   Bash
   ```
   python3.11 -m venv venv
   source venv/bin/activate
   ```

2. **Install the necessary SDK:**
   Bash
   ```
   pip install google-cloud-aiplatform
   ```

   This package contains the client libraries for interacting with the full suite of Vertex AI services, including its generative models.[10]

Authentication will be handled via Application Default Credentials (ADC). For local development, running gcloud auth application-default login in the terminal will open a browser window to authenticate with a user account.[3] The resulting credentials are stored locally and are automatically discovered by the

google-cloud-aiplatform library. When deployed to a Google Cloud environment (such as Cloud Run, Cloud Functions, or a Compute Engine VM), the SDK will automatically discover and use the credentials of the attached service account (sa-universal-clone-poc created via Terraform). This strategy is highly secure and portable, as it requires no service account keys or other credentials to be hardcoded or managed within the application code.

2.2 Streaming with PaLM 2: The text-bison Implementation

The text-bison model, part of the PaLM 2 family, is well-suited for single-turn, non-chat interactions like summarization or classification.[11] The SDK provides the

TextGenerationModel class, which includes a predict_streaming method to receive the model's output as it is generated. This is ideal for user-facing applications where perceived latency is a concern.

The following Python sample demonstrates how to call text-bison with streaming:

Python

```
# generate_bison.py
import vertexai
from vertexai.language_models import TextGenerationModel
import sys

def stream_text_bison(project_id: str, location: str, prompt: str) -> None:
    """Streams a response from the text-bison model."""

    # Initialize the Vertex AI SDK for Python
    # This must be done before any other SDK calls.
```

```
    vertexai.init(project=project_id, location=location)

    # Load the pretrained text-bison model
    # The 'text-bison' identifier points to the latest stable version.
    model = TextGenerationModel.from_pretrained("text-bison")

    # Define model parameters. These control the output's characteristics.
    parameters = {
        "temperature": 0.2,      # Controls randomness. Lower is more deterministic.
        "max_output_tokens": 512, # Maximum number of tokens in the response.
        "top_p": 0.95,           # Nucleus sampling.
        "top_k": 40,             # Top-k sampling.
    }

    print(f"--- Sending prompt to text-bison: '{prompt}' ---")

    # Call the predict_streaming method. This returns an iterator.
    responses = model.predict_streaming(prompt=prompt, **parameters)

    # Iterate through the stream and print each partial response.
    # In a real application, you would yield these chunks to a frontend.
    for response in responses:
        print(response.text, end="")

    print("\n--- Stream finished ---")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python generate_bison.py \"<your_prompt>\"")
        sys.exit(1)

    # Example usage from command line
    # Replace with your actual project ID and location.
    PROJECT_ID = "your-gcp-project-id"
    LOCATION = "us-central1" # text-bison is primarily available in us-central1

    user_prompt = sys.argv
    stream_text_bison(project_id=PROJECT_ID, location=LOCATION,
prompt=user_prompt)
```

This script initializes the SDK, loads the text-bison model, and calls predict_streaming. It then iterates over the response generator, printing each text chunk as it arrives.[13]

2.3 Future-Proofing with Gemini: The Recommended Path

While text-bison is capable, the Gemini family of models represents Google's latest and most advanced technology.[14] For any new project, building on the Gemini API is the architecturally sound choice. Models like

gemini-2.0-flash-001 or the even more cost-effective gemini-2.0-flash-lite-001 offer an excellent balance of performance and price for a wide range of tasks.[15]

The following sample demonstrates streaming with a Gemini model:

Python

```python
# generate_gemini.py
import vertexai
from vertexai.generative_models import GenerativeModel, Part
import sys

def stream_gemini(project_id: str, location: str, prompt: str) -> None:
    """Streams a response from a Gemini model."""

    # Initialize the Vertex AI SDK for Python
    vertexai.init(project=project_id, location=location)

    # Load the specified Gemini model.
    # gemini-2.0-flash-001 is a fast, multimodal, and cost-effective choice.
    model = GenerativeModel("gemini-2.0-flash-001")

    print(f"--- Sending prompt to Gemini: '{prompt}' ---")
```

```
    # The generate_content method with stream=True returns an iterator.
    responses = model.generate_content(
        [Part.from_text(prompt)],
        stream=True,
    )

    # The response structure for Gemini is slightly different.
    # We access the text from the 'parts' attribute of each candidate.
    for response in responses:
        # To prevent errors on the final empty response part
        if response.candidates and response.candidates.content.parts:
            print(response.candidates.content.parts.text, end="")

    print("\n--- Stream finished ---")

if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python generate_gemini.py \"<your_prompt>\"")
        sys.exit(1)

    # Example usage from command line
    # Replace with your actual project ID and location.
    # Gemini models support the 'global' location for higher availability.
    PROJECT_ID = "your-gcp-project-id"
    LOCATION = "global"

    user_prompt = sys.argv
    stream_gemini(project_id=PROJECT_ID, location=LOCATION, prompt=user_prompt)
```

This script uses the GenerativeModel class and the generate_content method with stream=True.[16] Note the slight difference in how the streamed text is accessed from the response object.

2.4 The SDK Transition and n8n Integration

The provided code samples highlight a clear evolution in the Vertex AI SDK. The

vertexai.language_models module used for PaLM/Bison is being superseded by the more versatile vertexai.generative_models module for Gemini.[13] While both are provided here for completeness, the strategic recommendation is to build all new features around the Gemini SDK. It supports a wider range of functionalities, including multimodality (image, video, and audio inputs), advanced function calling, and is the focus of Google's ongoing development and performance improvements.[14] Building on the older SDK introduces immediate technical debt.

To integrate these Python scripts into the specified n8n automation stack, the Execute Command node is the most direct method. A workflow can be designed where an n8n node passes a dynamic prompt to the script as a command-line argument.

- **n8n Node:** Execute Command
- **Command:** python3 /path/to/generate_gemini.py "{{$json.prompt}}"

The n8n expression {{$json.prompt}} would dynamically insert the prompt from a previous step in the workflow. The Python script's output, which is printed to standard output (stdout), can then be captured by the n8n node and passed to subsequent steps in the automation for further processing. This pattern provides a simple yet powerful bridge between the n8n orchestration layer and the Python-based AI logic.

Part 3: Financial Governance and Cost Control (Day 5)

Launching an MVP in seven days is only a success if it doesn't result in an unexpected and unmanaged bill. This section establishes the essential financial guardrails to effectively utilize the $300 in new customer credits and to architect a financially sustainable service for the long term.[18]

3.1 The POC Burn-Down Calculator: A Universal Formula

A primary challenge in forecasting costs is that different model families use different billing units: older models like text-bison are priced per 1,000 characters, while newer Gemini models are priced per 1,000 tokens.[18] A token is roughly equivalent to 4 characters of typical English text, but this is only an approximation. An effective cost

calculator must handle this distinction programmatically.

The following Python function provides a universal way to calculate the cost of a single API call, which can be logged alongside each request to track burn-down against the free credit.

Python

```python
# cost_calculator.py

# Pricing data based on public documentation for the us-central1 region.
# Prices are per 1 million units (characters or tokens).
# Source: [18, 20]
MODEL_PRICING = {
    "text-bison": {
        "input_cost_per_million": 0.25, # Billed as $0.00025 per 1k chars
        "output_cost_per_million": 0.25, # Billed as $0.00025 per 1k chars
        "unit": "characters"
    },
    "gemini-2.0-flash-lite-001": {
        "input_cost_per_million": 0.075,
        "output_cost_per_million": 0.30,
        "unit": "tokens"
    },
    "gemini-2.0-flash-001": {
        "input_cost_per_million": 0.10,
        "output_cost_per_million": 0.40,
        "unit": "tokens"
    },
    "gemini-1.5-pro": { # Using the <= 128k context window pricing
        "input_cost_per_million": 1.25,
        "output_cost_per_million": 5.00,
        "unit": "tokens"
    }
}

def calculate_cost(model_id: str, input_units: int, output_units: int) -> float:
    """
```

```
    Calculates the estimated cost of a single Vertex AI API call.

    Args:
        model_id: The identifier of the model used (e.g., 'text-bison', 'gemini-1.5-pro').
        input_units: The number of input characters or tokens.
        output_units: The number of output characters or tokens.

    Returns:
        The estimated cost in USD.
    """
    # Find the base model ID for pricing lookup
    base_model_id = next((key for key in MODEL_PRICING if key in model_id), None)

    if not base_model_id:
        raise ValueError(f"Pricing information not found for model: {model_id}")

    pricing_info = MODEL_PRICING[base_model_id]

    input_cost = (input_units / 1_000_000) * pricing_info["input_cost_per_million"]
    output_cost = (output_units / 1_000_000) * pricing_info["output_cost_per_million"]

    total_cost = input_cost + output_cost
    return total_cost

# Example Usage:
# For text-bison, assume 1000 input chars and 2000 output chars
cost_bison = calculate_cost("text-bison@002", 1000, 2000)
print(f"Estimated cost for text-bison call: ${cost_bison:.6f}")

# For Gemini, assume 500 input tokens and 1500 output tokens
cost_gemini = calculate_cost("gemini-1.5-pro", 500, 1500)
print(f"Estimated cost for Gemini Pro call: ${cost_gemini:.6f}")
```

To make informed, cost-based architectural decisions, the following table provides a clear comparison of model pricing. This data directly powers the calculator and highlights the significant cost disparities that serve as the primary lever for optimization.

Model Identifier	Family	Input Cost (per 1M units)	Output Cost (per 1M units)	Unit

text-bison@002	PaLM 2	$0.25	$0.25	Characters
gemini-2.0-flash-lite-001	Gemini	$0.075	$0.30	Tokens
gemini-2.0-flash-001	Gemini	$0.10	$0.40	Tokens
gemini-1.5-pro-002	Gemini	$1.25 (≤128k tokens)	$5.00 (≤128k tokens)	Tokens

3.2 Proactive Budgeting in Google Cloud

A proactive budget is the most effective defense against billing surprises. A budget should be configured in the Google Cloud Billing console to monitor the consumption of the $300 credit.

Configuration Steps:

1. Navigate to the **Billing** section in the Google Cloud Console.
2. Select **Budgets & alerts** from the navigation menu.
3. Click **Create budget**.
4. Name the budget (e.g., "POC-300-Credit-Alert").
5. Set the **Budget amount** to a specified amount, such as $300.
6. Under **Alert threshold rules**, configure alerts for when the actual cost reaches 50%, 90%, and 100% of the budget.
7. For the 100% alert, under **Manage notifications**, select **Connect a Pub/Sub topic to this budget**. This enables a powerful enterprise pattern where the budget alert can trigger a Cloud Function to programmatically disable billing for the project or shut down non-essential services, acting as an automated kill switch.

3.3 Total Cost of Ownership Perspective

While model inference costs will constitute the vast majority of the POC's expenses, a complete Total Cost of Ownership (TCO) view must consider all contributing services.[21] Minor charges can accrue from services like Cloud Storage (for storing

Terraform state files), Cloud Logging (for API call logs), and network egress.

Attempting to micromanage the cost of each of these minor supporting services is inefficient. The GCP Budget and alert system provides a holistic view by monitoring the *entire project bill*, not just Vertex AI usage. This makes it the most reliable and efficient governance tool. Relying on this comprehensive system is a more robust strategy than trying to track the cost of every individual component, ensuring that no source of expenditure, however small, goes unnoticed.

Part 4: Advanced Operations and Long-Term Optimization (Days 6-7)

The final phase of the 7-day sprint transitions from initial development to operational resilience and strategic planning. This involves preparing for real-world constraints like service quotas and architecting a path from the free-tier POC to a sustainable, ultra-low-cost production service.

4.1 Navigating Regional and Quota Constraints

Cloud platforms use quotas to ensure fair resource allocation and service stability.[22] These quotas can manifest as limits on requests per minute (RPM) or tokens per minute (TPM).[23] Research reveals a significant operational advantage for newer models: many Gemini models use a Dynamic Shared Quota (DSQ) system, which dynamically distributes capacity among users, reducing the likelihood of hitting a hard limit and removing the need for manual quota increase requests. Older models often rely on stricter, fixed regional quotas.[24]

Primary Strategy: The global Endpoint
The single most effective strategy for mitigating regional capacity issues and quota-related errors is to use the global endpoint, which is supported for many modern Gemini models.25 This endpoint allows Google's infrastructure to route an API request to any data center worldwide that has available capacity, dramatically improving availability and reducing the chance of receiving a

429 ResourceExhausted error.

To implement this, the Python SDK initialization should be modified as follows:
vertexai.init(project=PROJECT_ID, location='global')
This should be the default configuration for any application using a supported Gemini model.

Contingency Plan: Regional Failover
For models that do not support the global endpoint (such as older PaLM models or certain specialized models) or in the rare event of a widespread capacity issue, a regional failover mechanism is a prudent contingency plan. This involves programmatically retrying a failed request in a different region.
The following Python pseudo-code illustrates this pattern:

Python

```
# A list of viable regions for the model in use.
# This should be populated based on the model's documentation.
REGIONS = ["us-central1", "europe-west4", "asia-southeast1"]

def call_with_failover(prompt):
    for region in REGIONS:
        try:
            print(f"Attempting call in region: {region}")
            vertexai.init(project=PROJECT_ID, location=region)
            model = GenerativeModel("model-without-global-support")
            response = model.generate_content(prompt)
            return response
        except google.api_core.exceptions.ResourceExhausted as e:
            print(f"Resource exhausted in {region}. Trying next region. Error: {e}")
            continue
    raise Exception("All regions failed due to resource exhaustion.")
```

This strategy requires knowledge of which regions support the target model. The following table provides a high-level overview to inform this logic.

Model Family	Key Regions	Global Endpoint Support

text-bison (PaLM 2)	us-central1 (primarily) [26]	No
gemini-1.5-pro	us-central1, europe-west4, asia-southeast1 [25]	No
gemini-2.0-flash	us-central1, europe-west4, asia-southeast1 [25]	Yes
gemini-2.0-flash-lite	us-central1, europe-west4, asia-southeast1 [25]	Yes

4.2 Post-Credit Strategy: Achieving a Sub-$5/Month Bill

Operating a generative AI service for less than $5 per month is an extreme-frugality goal that demands deliberate and aggressive optimization. It cannot be achieved by accident. The following three architectural patterns are not mere suggestions but are requirements for meeting this target after the free credits are exhausted.

Trick 1: Aggressive Model Tiering
Not all tasks require the same level of model intelligence.27 A significant cost-saving measure is to implement application-level routing that directs requests to the most cost-effective model capable of performing the task.

- **Concept:** For simple, low-stakes tasks like basic text reformatting, simple classification, or keyword extraction, the request should be routed to the absolute cheapest model available, such as gemini-2.0-flash-lite-001. Only for tasks requiring complex reasoning, nuanced creativity, or deep analysis should the logic escalate to a more powerful and expensive model like gemini-1.5-pro.
- **Implementation:** The application should include logic, potentially a simple rules engine or a classification model itself, to assess the complexity of an incoming prompt and select the appropriate model ID before making the API call.

Trick 2: Implementing a Semantic Caching Layer
Repeatedly calling the API with the same or semantically similar prompts is a primary source of wasted expenditure.28 A caching layer can dramatically reduce the number of API calls.

- **Concept:** Before calling the Vertex AI API, the application should first check a cache. For a simple implementation, the cache key can be a hash of the exact prompt string. For a more advanced "semantic" cache, the application can generate an embedding of the incoming prompt and check for cached results

from other prompts with a high cosine similarity.

- **Implementation:** This can be built using a low-cost, in-memory key-value store like Google Cloud Memorystore (Redis) or a serverless database. The application logic would be: 1) Receive prompt. 2) Check cache for prompt. 3) If hit, return cached response. 4) If miss, call Vertex AI API. 5) Store the new prompt and its response in the cache. 6) Return response.

Trick 3: Token-Aware Prompt Engineering and Strict Output Control
The cost of an API call is a direct function of the number of input and output tokens.30 Therefore, minimizing both is a direct cost optimization strategy.27

- **Implementation:**
  1. **Prompt Minification:** Engineer prompts to be as concise as possible while retaining clarity. For example, instead of a verbose request like, "Could you please take the following text and summarize it for me into three main bullet points?", a more token-efficient prompt would be, "Summarize in 3 bullets: [text]". This directly reduces input token count and cost.
  2. **Output Gating:** Utilize the stop_sequences parameter in the API call. If the desired output is a JSON object, for instance, setting the closing brace } as a stop sequence can prevent the model from generating extraneous conversational text like "Here is the JSON you requested:", thereby saving on output tokens.
  3. **Advanced Context Caching:** For applications that repeatedly reference the same large documents (e.g., a chatbot answering questions about a specific user manual), Google's native Context Caching feature can be explored. This allows frequently used context to be cached on the server side, reducing the number of tokens that need to be sent with each request and lowering costs.[21]

Conclusion: The Path from POC to Production

This seven-day sprint provides the blueprint for launching a functional, secure, and financially governed Generative AI MVP. The resulting POC is not a throwaway experiment but the first iteration of a production-ready system, built on a solid foundation of Infrastructure-as-Code and enterprise best practices.

The logical evolution from this POC involves several key steps. The simple Python scripts can be containerized and deployed to a scalable, serverless platform like Cloud Run or Cloud Functions for higher availability and easier management. The MLOps

process can be formalized using Vertex AI Pipelines to automate model evaluation, testing, and deployment, creating a more robust and repeatable workflow.[29] Finally, to enhance the "Universal Clone" platform's capabilities, the next architectural step would be to explore Retrieval-Augmented Generation (RAG). By integrating with Vertex AI Search, the models can be grounded in proprietary data sources, allowing them to provide contextually aware and factually accurate responses based on specific business information, transforming the generic models into specialized, high-value assets.[27] This forward-looking path ensures that the initial investment in this 7-day sprint serves as a strategic launchpad for a sophisticated and scalable AI-powered platform.

Cytowane prace

1. Provision Vertex AI Workbench resources with Terraform | Google Cloud, otwierano: czerwca 16, 2025, https://cloud.google.com/vertex-ai/docs/workbench/instances/terraform
2. Terraform support for Vertex AI | Google Cloud, otwierano: czerwca 16, 2025, https://cloud.google.com/vertex-ai/docs/start/use-terraform-vertex-ai
3. Quickstart: Generate text using the Vertex AI Gemini API - Google Cloud, otwierano: czerwca 16, 2025, https://cloud.google.com/vertex-ai/generative-ai/docs/start/quickstarts/quickstart-multimodal
4. google_project_service | Resources | hashicorp/google - Terraform Registry, otwierano: czerwca 16, 2025, https://registry.terraform.io/providers/hashicorp/google/4.27.0/docs/resources/google_project_service
5. google_project_service | Resources | hashicorp/google | Terraform ..., otwierano: czerwca 16, 2025, https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/google_project_service
6. google_service_account | Resources | hashicorp/google - Terraform Registry, otwierano: czerwca 16, 2025, https://registry.terraform.io/providers/hashicorp/google/3.24.0/docs/resources/google_service_account
7. google_service_account | Resources | hashicorp/google | Terraform ..., otwierano: czerwca 16, 2025, https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/google_service_account
8. hashicorp/google - IAM policy for projects - Terraform Registry, otwierano: czerwca 16, 2025, https://registry.terraform.io/providers/hashicorp/google/latest/docs/resources/google_project_iam
9. Vertex AI access control with IAM | Google Cloud, otwierano: czerwca 16, 2025, https://cloud.google.com/vertex-ai/docs/general/access-control

10. Setting up Vertex AI SDK for Python - Stack Overflow, otwierano: czerwca 16, 2025, https://stackoverflow.com/questions/77302196/setting-up-vertex-ai-sdk-for-python
11. PaLM 2 Text Bison – Vertex AI - Google Cloud Console, otwierano: czerwca 16, 2025, https://console.cloud.google.com/vertex-ai/publishers/google/model-garden/text-bison?hl=es
12. PaLM 2 Text Bison – Vertex AI - Google Cloud Console, otwierano: czerwca 16, 2025, https://console.cloud.google.com/vertex-ai/publishers/google/model-garden/text-bison?hl=de&inv=1&invt=AbeE3g
13. vertex-ai-samples/notebooks/official/prediction ... - GitHub, otwierano: czerwca 16, 2025, https://github.com/GoogleCloudPlatform/vertex-ai-samples/blob/main/notebooks/official/prediction/llm_streaming_prediction.ipynb
14. Getting Started with the Vertex AI Gemini API and Python SDK GSP1209 - YouTube, otwierano: czerwca 16, 2025, https://www.youtube.com/watch?v=U8uOPhpSm1I
15. Google models | Generative AI on Vertex AI, otwierano: czerwca 16, 2025, https://cloud.google.com/vertex-ai/generative-ai/docs/models
16. generative-ai/gemini/getting-started/intro_gemini_express.ipynb at main - GitHub, otwierano: czerwca 16, 2025, https://github.com/GoogleCloudPlatform/generative-ai/blob/main/gemini/getting-started/intro_gemini_express.ipynb
17. Gemini API libraries | Google AI for Developers, otwierano: czerwca 16, 2025, https://ai.google.dev/gemini-api/docs/libraries
18. Google Cloud Vertex AI Pricing Review 2025: Plans & Costs - Tekpon, otwierano: czerwca 16, 2025, https://tekpon.com/software/google-cloud-vertex-ai/pricing/
19. Vertex AI – Google Cloud console, otwierano: czerwca 16, 2025, https://console.cloud.google.com/vertex-ai/publishers/google/model-garden/gemma3
20. Gemini Developer API Pricing | Gemini API | Google AI for Developers, otwierano: czerwca 16, 2025, https://ai.google.dev/gemini-api/docs/pricing
21. Optimizing AI costs: Three proven strategies | Google Cloud Blog, otwierano: czerwca 16, 2025, https://cloud.google.com/transform/three-proven-strategies-for-optimizing-ai-costs
22. Vertex AI quotas and limits - Google Cloud, otwierano: czerwca 16, 2025, https://cloud.google.com/vertex-ai/docs/quotas
23. Rate limits and quotas | Firebase AI Logic - Google, otwierano: czerwca 16, 2025, https://firebase.google.com/docs/ai-logic/quotas
24. Generative AI on Vertex AI quotas and system limits - Google Cloud, otwierano: czerwca 16, 2025, https://cloud.google.com/vertex-ai/generative-ai/docs/quotas
25. Deployments and endpoints | Generative AI on Vertex AI - Google Cloud,

otwierano: czerwca 16, 2025, https://cloud.google.com/vertex-ai/generative-ai/docs/learn/locations
26. Re: Locations Available For Text-Bison Other Than us-central1 - Google Cloud Community, otwierano: czerwca 16, 2025, https://www.googlecloudcommunity.com/gc/AI-ML/Locations-Available-For-Text-Bison-Other-Than-us-central1/m-p/635228
27. Generative AI Cost Optimization Strategies | AWS Cloud Enterprise Strategy Blog, otwierano: czerwca 16, 2025, https://aws.amazon.com/blogs/enterprise-strategy/generative-ai-cost-optimization-strategies/
28. Re: Confused about pricing differences between Vertex AI and Google AI Studio, otwierano: czerwca 16, 2025, https://www.googlecloudcommunity.com/gc/AI-ML/Confused-about-pricing-differences-between-Vertex-AI-and-Google/m-p/887859
29. Cost Optimization Tips for Running AI in Google Cloud - Visualpath, otwierano: czerwca 16, 2025, https://visualpathblogs.com/google-cloud-ai/cost-optimization-tips-for-running-ai-in-google-cloud/
30. Vertex AI Pricing: A Comprehensive Guide for Tech Professionals and Businesses, otwierano: czerwca 16, 2025, https://www.byteplus.com/en/topic/447868
31. Better, faster, more affordable: Google Cloud significantly expands Vertex AI portfolio, otwierano: czerwca 16, 2025, https://www.techzine.eu/news/applications/121711/better-faster-cheaper-google-cloud-significantly-expands-vertex-ai-portfolio/